CYBIL for NOS/VE Keyed-File and Sort/Merge Interfaces

CONTROL DATA

Usage

60464117

CYBIL for NOS/VE Keyed-File and Sort/Merge Interfaces

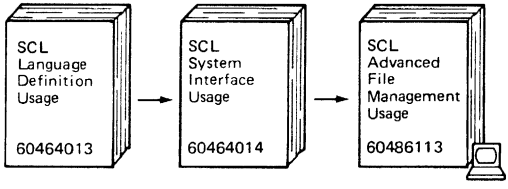
Usage

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

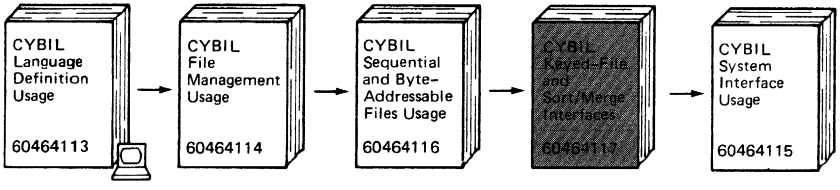
Publication Number 60464117

Related Manuals

Background (Access as Needed):



CYBIL Manual Set:



Additional References:

→ Indicates the reading sequence.

Indicates an online version of the manual is available.

© 1985 by Control Data Corporation.
All rights reserved.
Printed in the United States of America.

Manual History

This revision:

Revision B documents the CYBIL interfaces to AAM 1.1 and Sort/Merge 1.1 for NOS/VE Version 1.1.3 at PSR level 644. It was printed in October, 1985.

This revision documents the new keyed-file interface features: nested files, write concurrency (locks), and direct-access file organization, and the new Sort/Merge features: individual procedure declaration decks and indexed-sequential file support.

Previous Revision	System Version/ PSR Level	Product Version	Published Date
A	1.1.2/630	1.0	March 1985

Contents

About This Manual 7

Introduction Introduction-1

Part I. Keyed-File Interface

Keyed-File Concepts I-1-1
Using the CYBIL Keyed-File Interface I-2-1
Keyed-File Interface Calls I-3-1
Keyed-File Attributes I-4-1

Part II. Sort/Merge Interface

Introduction to Sort/Merge II-1-1
Sort/Merge Procedure Calls II-2-1
Owncode Procedures II-3-1

Appendixes

Glossary A-1
ASCII Character Set B-1
Constant and Type Declarations C-1
Collation Tables D-1
Common Procedures E-1

Index Index-1

About This Manual

This manual describes CONTROL DATA® CYBIL procedure calls that serve as the interface between the CDC® Network Operating System/Virtual Environment (NOS/VE) and CYBIL programs. CYBIL is the implementation language for NOS/VE.

The CYBIL program interface is described in these manuals:

CYBIL File Management
CYBIL Sequential and Byte Addressable Files
CYBIL Keyed-File and Sort/Merge Interfaces
CYBIL System Interface

This manual, CYBIL Keyed-File and Sort/Merge Interfaces Usage, describes the interfaces that allow CYBIL programs to use keyed files and the Sort/Merge package.

Audience

This manual is a reference for CYBIL programmers. It assumes that the reader knows the CYBIL programming language as described in the CYBIL Language Definition manual.

To use the procedure calls described in this manual, the programmer must copy decks from a system source library. Although the manual introduction provides a brief description of the commands required to copy decks, the complete description is in the SCL Source Code Management manual.

This manual also assumes that the reader is familiar with the NOS/VE command interface, the System Command Language (SCL). All commands referenced in this manual are SCL commands. The SCL command syntax is described in the SCL Language Definition manual; SCL commands are described in the SCL System Interface and SCL Advanced File Management manuals.

CYBIL Manual Set

This manual belongs to the CYBIL manual set. Besides this manual, the CYBIL manual set is composed of these manuals:

CYBIL Language Definition

Contains the complete language specification for CYBIL, the NOS/VE implementation language, and an explaination of the Debug utility as used with CYBIL.

CYBIL File Management

Describes the procedure calls that interface between a CYBIL program and the NOS/VE file system. It describes local file management and the assignment of files to device classes with a chapter describing each device class. It also describes file attribute definition and file opening and closing.

CYBIL Sequential and Byte Addressable Files

Describes the procedure calls that allow a CYBIL program to read and write sequential and byte addressable files. It describes both segment access and record access.

CYBIL System Interface

Describes system-defined CYBIL procedures that serve as the interface between a program and non-I/O system capabilities. It describes program management, condition processing, interstate communication, and system command language (SCL) calls.

Manual Organization

This manual, CYBIL Keyed-File and Sort/Merge Interfaces, contains:

- An introduction that applies to both part I and part II
- Part I describing the keyed-file interface
- Part II describing the Sort/Merge interface
- Appendixes including:
  - Glossary
  - ASCII character set listing
  - Alphabetical listing of CYBIL constant and type declarations
  - Description of how to create and use collation tables and listings of the NOS/VE predefined collation tables.
  - Source listings of the CYBIL procedures used to report status in the example programs

Conventions

This manual uses these conventions:

boldface	Denotes the required parts of a format.
italics	Denotes the optional parts of a format.
blue	Denotes user input within interactive session examples.
UPPERCASE	In formats, it denotes the parts of the format that must be entered exactly as shown. In text, names and identifiers are shown in uppercase.
lowercase	In formats, it denotes the parts of the format that the user supplies.
`nonproportional typeface`	Denotes examples (the nonproportional typeface simulates computer output). User input is indicated by blue print, system output by black print.
number base	All numbers are decimal unless otherwise indicated.

▮	Vertical bars in the margin indicate changes or additions to the text from the previous revision.
•	A dot next to the page number indicates that a significant amount of text (or the entire page) has changed from the previous revision.

Ordering Manuals

Control Data manuals are available through Control Data sales offices or through:

Control Data Corporation
Literature Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Submitting Comments

The last page of this manual is a comment sheet. Please tell us about any errors you found in this manual and any problems you had using it.

If the comment sheet in this manual has been used, please send your comments to:

Control Data Corporation
Publications and Graphics Division
P.O. Box 3492
Sunnyvale, California 94088-3492

Please include this information with your comments:

▮ The manual title, publication number, and revision level (for this manual: CYBIL Keyed-File and Sort/Merge Interfaces Usage, 60464117 B)

Your system's PSR level (if you know it)

Your name, your company's name and address, your work phone number, and whether you want a reply

Also, if you have access to SOLVER, the CDC online facility for reporting problems, you can use it to submit comments about this manual. When it prompts you for a product identifier for your report, please specify AA8 when commenting on the keyed-file interface and SM8 when commenting on the Sort/Merge interface.

How to Use CYBIL Program Interface Calls

Copying Procedure Decks Into Your Program 1
Procedure Deck Names 2
Expanding Your Program 2
Executing Your Program 2.1

Procedure Calls in Your Program 3

Status Checking 4
Exception Condition Information 4

System Naming Convention 6

How to Use CYBIL Program Interface Calls

NOS/VE provides a set of CYBIL procedures, called the program interface, by which programs can request system services. This manual describes two parts of the program interface: the keyed-file interface and the Sort/Merge interface. The rest of the program interface is described in the CYBIL File Management, CYBIL Sequential and Byte Addressable Files, and CYBIL System Interface manuals.

Copying Procedure Decks Into Your Program

The CYBIL procedure declarations for the CYBIL program interface procedures reside in decks in a system source library. To use a program interface procedure, you copy the text from the appropriate decks into the source text of your program. (This process is described in detail in the SCL Source Code Management manual.)

To copy deck text into your program, embed *COPYC Source Code Utility (SCU) directives into your program. Each directive is a separate line and the directive must begin in column one. The directive specifies the name of a deck to be inserted at that point in the text. For example, the following directive requests insertion of the AMP$OPEN deck:

```
*COPYC AMP$OPEN
```

The deck text is inserted in your program when you execute the Source Code Utility (SCU) to process the embedded directives as it expands your program.

It is suggested that you embed the *COPYC directives between the PROCEND and MODEND statements at the end of your program. This is so that line numbers returned by CYBIL runtime error message do not include the inserted procedure declaration text. A line number that includes the inserted text is less useful. For example, if the procedure declarations were inserted at the beginning of your source code, a message referencing line number 1270 might refer to line 42 of your source code.

Procedure Deck Names

To use CYBIL program interface calls, you copy a deck for each procedure call you use. The deck has the same name as the procedure call.

For example, if your program uses the AMP$OPEN, AMP$GET_KEY, and AMP$CLOSE calls, it must use these three directives:

```
*COPYC AMP$OPEN
*COPYC AMP$GET_KEY
*COPYC AMP$CLOSE
```

Expanding Your Program

Before you compile a CYBIL program that uses program interface calls, you use SCU to expand the program, as follows:

1. You must begin with an existing source library file. If you do not have one, you can create an empty source library using the CREATE_SOURCE_LIBRARY command.
2. Start an SCU utility session, specifying a source library file.
3. Create one or more decks containing your program text.
4. Expand the decks containing your program text. Specify these two files as the alternate base libraries from which SCU copies the program interface decks:

   $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE
   $SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE
5. End the SCU utility session.

This process gives you the expanded program text that can be compiled.

The following is a minimal command sequence that performs the preceding steps (numbered 1 through 5). It uses only temporary files and assumes your program text is on file $USER.PROGRAM_TEXT. (/, sc/, and sc../ are system prompts; you do not enter them.)

1. `/create_source_library result=temporary_library`
2. `/scu base=temporary_library`
3. ```
   sc/create_deck deck=temporary_deck ..
   sc../modification=temporary_modification source=$user.program_text
   ```
4. ```
   sc/expand_deck deck=temporary_deck ..
   sc../alternate_base=($system.cybil.osf$program_interface, ..
   sc../$system.common.psf$external_interface_source)
   ```
5. `sc/quit write_library=no`

The EXPAND_DECK subcommand writes the expanded program text on file COMPILE. You could next compile the expanded program text with a command such as this:

```
/cybil input=compile list=listing list_options=(r, a)
```

The CYBIL command is described in the CYBIL Language Definition manual. For more information on source libraries and source text expansion, see the SCL Source Code Management manual.

Executing Your Program

When the compiled program is a CYBIL program containing any of the calls described in this manual, you must add an object library to the program library list before executing the program. The object library file to be added to the list is as follows:

Keyed-file interface calls: $LOCAL.AAF$44D_LIBRARY

Sort/Merge interface calls: $LOCAL.SMF$LIBRARY

This step is required so that modules can be loaded from the object libraries.

The commands that can add an object library to the program library list are described in the SCL Object Code Management manual. (If program execution is initiated within another CYBIL program, a CYBIL call can add the required object library to the program library list as described in the CYBIL System Interface manual.)

For example, the following SET_PROGRAM_ATTRIBUTES command adds both object libraries to the program library list; the LGO command executes the object modules on file LGO:

```
set_program_attributes, ..
  add_libraries=($local.aaf$44d_library, $local.smf$library)
lgo
```

The following EXECUTE_TASK command performs the same operations as the preceding two commands:

```
execute_task, file=lgo, ..
  libraries=($local.aaf$44d_library, $local.smf$library)
```

Procedure Calls in Your Program

A call to a program interface procedure has the same format as any other CYBIL procedure call. It consists of the procedure name followed by a parameter list enclosed in parentheses and terminated by a semicolon. For example, this is a call to open a file:

```
AMP$OPEN ( lfn, AMC$RECORD_ACCESS, NIL, fid, status );
```

NOTE

You cannot omit parameters in a procedure call. You must specify a value (or a variable containing an appropriate value) for each parameter in the procedure call format. The parameter values must be specified in the order shown in the call format.

The CYBIL compiler performs type checking on all parameter values. The type of each parameter value must conform to the type specified for the parameter in the procedure declaration.

The parameter type is given in the parameter description. For example, consider this parameter description:

status: VAR of ost$status

Status variable in which the completion status is returned.

This parameter description describes the status parameter. The words VAR OF indicate that it is a reference parameter, meaning that the procedure returns a value to the caller in the specified variable. The parameter type is OST$STATUS.

The CYBIL type declarations for the calls described in this manual are listed in alphabetical order in appendix C.

Status Checking

The last parameter on every program interface call is the status parameter. You must specify a status variable (type OST$STATUS) as the last parameter on a call. When the procedure completes, it returns its completion status in the specified status variable.

You can specify an error-exit procedure to process errors returned by file interface procedures. (It does not process Sort/Merge errors.) The error-exit procedure is specified by the error_exit_name or error_exit_procedure file attribute.

If an error-exit procedure is specified for an instance of open, a file interface procedure calls the error-exit procedure when it returns abnormal status. The abnormal status is passed to the error-exit procedure which, in turn, passes its completion status to the status variable specified on the call.

An error-exit procedure is effective only while the file is open. It is not effective for AMP$OPEN or AMP$CLOSE calls. For these calls, and for files without error-exit procedures, you must check the contents of the status variable after the call to determine if the call completed successfully.

A status record is returned in the status variable. If the NORMAL field of the status record is TRUE, the procedure completed normally. If the NORMAL field is FALSE, the procedure completed abnormally.

For example, these lines show an AMP$OPEN call and the status check following the call:

```
AMP$OPEN ( lfn, AMC$RECORD_ACCESS, NIL, fid, status );
IF NOT status.NORMAL THEN
   PMP$EXIT( status );
IFEND;
```

For the PMP$EXIT call description and additional information on condition handling, see the CYBIL System Interface manual. A more complete example of status variable processing is given by the p#inspect_status_variable and p#display_status_variable procedures in appendix E.

Exception Condition Information

When the procedure completes abnormally, the procedure returns additional information about the exception condition (the error) that occurred. The following variant fields of the OST$STATUS record return condition information when the key field, NORMAL, is FALSE:

IDENTIFIER

Two-character string identifying the process that detected the error. These are the process identifiers that could be returned by calls described in this manual:

AA	Keyed-file interface (Advanced Access)
AM	Access Method (lower-level input/output procedure called by the keyed-file interface)
OS	Operating System
SM	Sort/Merge
PF	Permanent File management
PM	Program management

CONDITION

Code that uniquely identifies the exception condition (an integer of type OST$STATUS_CONDITION). Your program should reference the exception condition by its condition identifier. (For example, AAE$KEY_NOT_FOUND is a keyed-file interface condition identifier.)

Each procedure description lists the condition identifiers of exception conditions commonly returned by the procedure; the list does not include all conditions that the procedure can return.

TEXT

Additional information about the condition contained in a string record of type OST$STRING. The record has two fields:

SIZE	The string length in characters (0 through 256)
VALUE	The text string

NOTE

The TEXT field does not contain the error message. It contains items of information that are inserted into the message template for the exception condition when an error message is formatted. For more information on message formatting, see the CYBIL System Interface manual.

The error-exit procedure or your program can also fetch the error severity level for an exception condition using an OSP$GET_STATUS_SEVERITY call (as described in the CYBIL System Interface manual).

System Naming Convention

In general, all CYBIL program interface identifiers follow a system naming convention as follows:

idx$name

id — Two characters identifying the process that uses the identifier. (These are the same process identifiers returned in the IDENTIFIER field of the status record.)

x — Character indicating the type of CYBIL element identified. These are the element types:

c	Constant
d	Declaration of multiple or complex types
e	Error condition
f	File
i	Inline text or code
k	Keypoint or keyword
m	Module
p	Procedure
s	Section
t	Type
v	Variable
x	Element with XDCL attribute

$ — The $ character indicates that CDC defined the identifier.

NOTE

To avoid redefining a CDC identifier, do not use the $ character in identifiers that you define.

name — A string describing the purpose of the element referenced by the identifier.

For example, the identifier AMP$CREATE_KEY_DEFINITION follows the naming convention:

- Its process identifier is AM (Access Method).
- It identifies a procedure (P).
- It is a CDC-defined identifier ($).
- Its purpose is the creation of an alternate-key definition (CREATE_KEY_DEFINITION)

Keyed-File Concepts I-1

Keyed-File Organizations I-1-1
Indexed-Sequential File Organization I-1-2
Indexed-Sequential File Structure I-1-2
Data-Block Split I-1-4
Index Levels I-1-6
Indexed-Sequential Primary Keys I-1-9
Direct-Access File Organization I-1-10
Direct-Access File Structure I-1-10
Hashing Procedure I-1-13
Direct-Access Primary Keys I-1-14
Alternate Keys I-1-15
Alternate-Key Characteristics I-1-15
The Alternate Index I-1-16
Alternate-Key Definition I-1-16
Duplicate Key Values I-1-17
Null Suppression I-1-19
Sparse-Key Control I-1-20
Concatenated Keys I-1-21
Repeating Groups I-1-22
Nested Files I-1-24

Keyed-File Concepts I-1

The CYBIL keyed-file interface is a group of procedure calls that perform operations on keyed files. A keyed file is a file whose file organization allows record access by key value.

Keyed files are like sequential and byte-addressable files in that the data in the files is contained in records.

A record is a collection of data that is read and written as a unit. The record could contain several fields of data, some of which have a fixed length while others vary in length. Thus, the records as a whole could have a fixed length or be variable in length.

For example, a record could contain three data items of different types: an integer, a floating point number, and a string of characters. To write a record, a program writes all three data items together as a record; when the record is later read, all three data items are delivered to the program.

The records in a sequential or byte-addressable file are stored as a simple sequence. The records in a keyed file are stored within a file structure as described in the following sections.

Keyed-File Organizations

A file is a keyed file if its file_organization attribute is either indexed-sequential or direct-access. A keyed-file organization allows you to read any record in the file directly by specifying its key value. The key value for a record is determined when the record is written to the file.

To allow you to access each record by a key value, the file organization must relate each key value to the location of the record in the file. The keyed-file interface performs all processing required to relate a key value to a record location; the user does not specify how this is done beyond choosing the file organization. The method of relating a key value to a record location differs for each keyed-file organization as described in the following sections.

Indexed-Sequential File Organization

The indexed-sequential file organization allows content addressing of records; that is, you can directly access a record by the contents of one or more fields of data in the record. The fields of data by which a record is addressed are its key fields, and the contents of those fields are its key values.

An indexed-sequential file always has a primary key. (It can also have one or more alternate keys as described in the Alternate Keys section of this chapter.)

Each primary-key value is unique within the file; there can be no duplicate primary-key values in a file.

The indexed-sequential file organization is used only when you can assign a unique value to each record stored in the file. This unique value is usually a field of data within the record (an embedded key), although it can be a value assigned to the record and not included in the record data (a nonembedded key).

For example, the primary key for an employee file could be the employee's name. However, because two employees could have the same name, it is better to assign a unique identification number to each employee and use that number as the primary key for the file.

The indexed-sequential file organization should be used if a requirement exists to read file records both sequentially and randomly. For example, the records in an employee file could be read sequentially to produce a listing of all employees or read randomly to update individual records.

When an indexed-sequential file is read sequentially, its records are accessed in ascending order by key value. The order is kept even when new records are added to the file. For example, if an employee file is read sequentially using its primary key (the employee identification number), the records are read in ascending order by their identification number.

Indexed-Sequential File Structure

This section gives a general description of the indexed-sequential structure. You can use indexed-sequential files without knowing their structure. However, if you understand the indexed-sequential structure and how it grows, you can create more efficient indexed-sequential files by specifying appropriate values for structural parameters.

The internal structure of an indexed-sequential file is designed to provide both random and sequential access to the data records in the file. File space is divided into blocks, all the same size.

A block contains a block header and one of the following:

Internal tables
Data records (a data block)
Index records (an index block)

Each index record points to a data block. The index record contains the location of the data block and the range of key values of the data records stored in that block.

You can display the contents of all components of an indexed-sequential file, the internal tables and index blocks as well as the data blocks, using the DISPLAY_KEYED_FILE command described in the SCL Advanced File Management Usage manual.

As you might expect, the actual internal index mechanism is complex. The simplified examples in this section, however, provide the level of detail you need to know in order to use indexed-sequential files.

To see how an index works, let's look at a very small file that contains one index block and two data blocks. As shown in figure I-1-1, the index block contains two index records. (The index records contain key values 1 and 5.) Each index record points to a data block in the file.

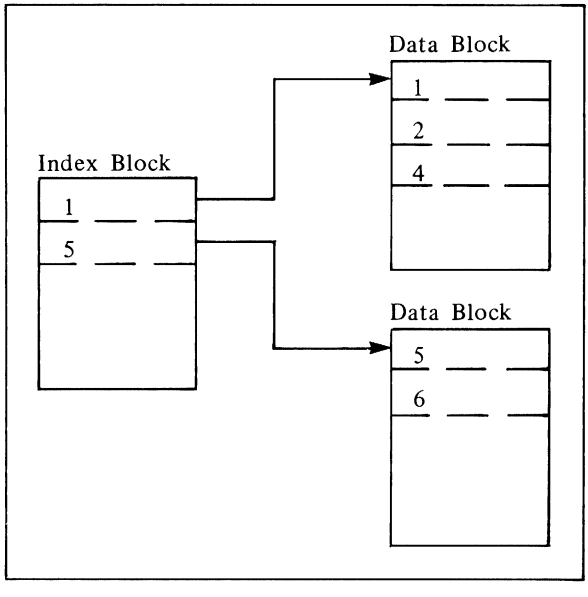


Figure I-1-1. Minimal Indexed-Sequential Structure

Let's suppose you request to read randomly the record with key value 6. When the record is read, these steps are performed:

1. The index records are searched to find the index record whose range of key values includes the key value 6.
2. After the correct index record (the second one) is found, the search for the record continues with the data block to which the second index record points.
3. The second data block is searched for the record with key value 6. When the record is found, its data is returned to the requestor.

Next, suppose you request that all records in the file shown in figure I-1-1 be read sequentially. These steps are performed:

1. The first index record is read to find the first data block.
2. The records from the first data block are read in order.
3. The second index record is read to find the second data block.
4. The records from the second data block are read in order.
5. The sequential read ends because there are no more index records and, so, no more data blocks to read.

This process reads the records in key-value order because both the index records and the data records are kept in key-value order.

Data-Block Split

Usually, a block has some empty space, called padding, that was left empty so that additional records could be written later to the block. Suppose, as shown in figure I-1-2, that a data block has been filled, a new record is to be written, and its key value is within the range of key values of the records in the full data block. For the file structure to be maintained, the data block must be split.

When a data-block split occurs, records in the data block whose key values are less than the key value of the new record remain in the existing block. All records in the existing block that come after the new record are moved to the newly created block.

The new record is put into either the new block or the existing block, depending on the relative amount of empty space in the blocks and the size of the new record. If the new record does not fit in either block, another new block is created and the new record is put into that block.

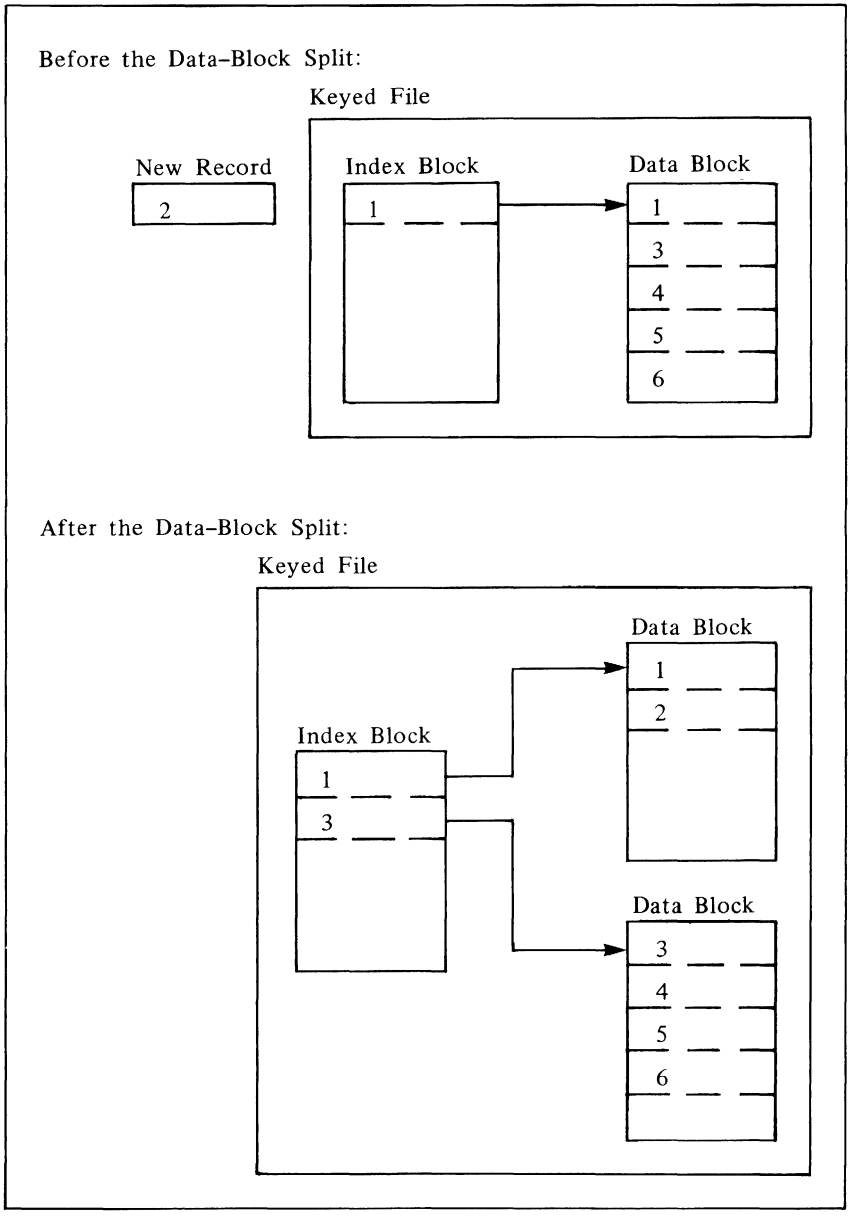


Figure I-1-2. Data-Block Split

Index Levels

As with data blocks, index blocks are also initially created with some empty space (index-block padding). However, for each new data block created due to a data-block split, another index record must be created. With the addition of many data records, the initial index block becomes full. When the index block is full, the next data-block split causes an index-block split.

As shown in figure I-1-3, when the initial index block splits, it causes the creation of another index level.

The index levels are numbered from the top down as index level 0, index level 1, and so forth. Index level 0 always has only one index block; it is always the starting point for an index search.

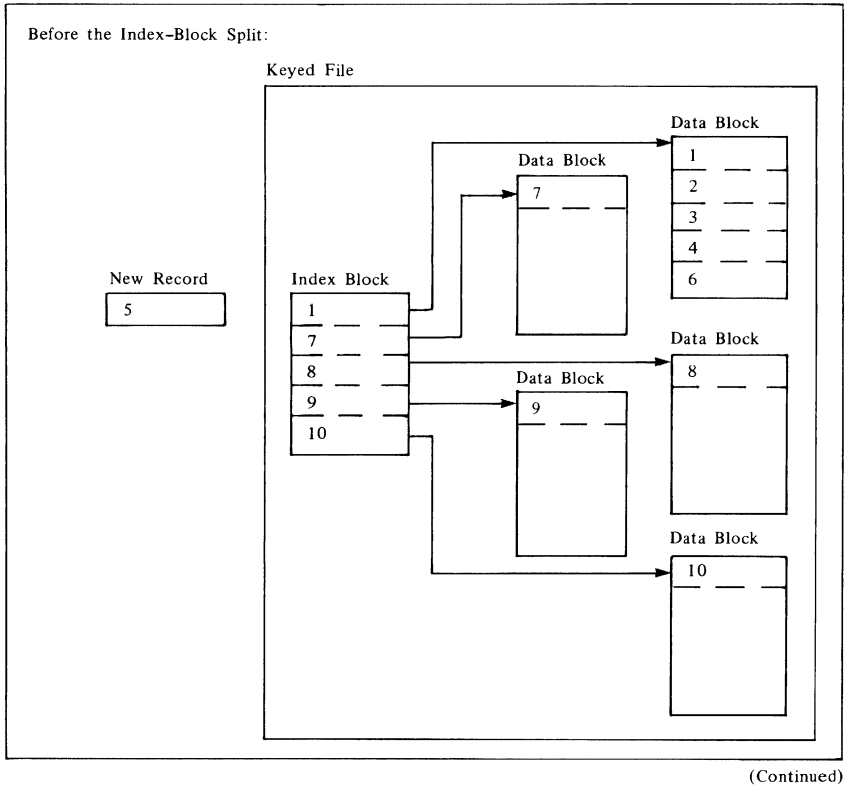
The index block at an upper level contains an index record for each index block at the next lower level. For example, the index block at level 0 contains an index record for each index block at level 1.

A search for a data record requires an index-block search at each index level. For example, the level-0 search finds the index record that points to the appropriate level-1 index block. If the file has only two index levels, the level 1 search finds the index record that points to the appropriate data block.

As you can see, the addition of another index level increases the time required to find an individual data record.

Index levels can be added up to the index-level limit of 15 levels. This sets a limit on the number of records in the file.

The index-level limit is reached when addition of another record to the file would require creation of another index level, but 15 index levels already exist in the file. When this happens, the index-level-overflow flag is set and no more records can be added to the file.



(Continued)

Figure I-1-3. Index-Block Split

(Continued)

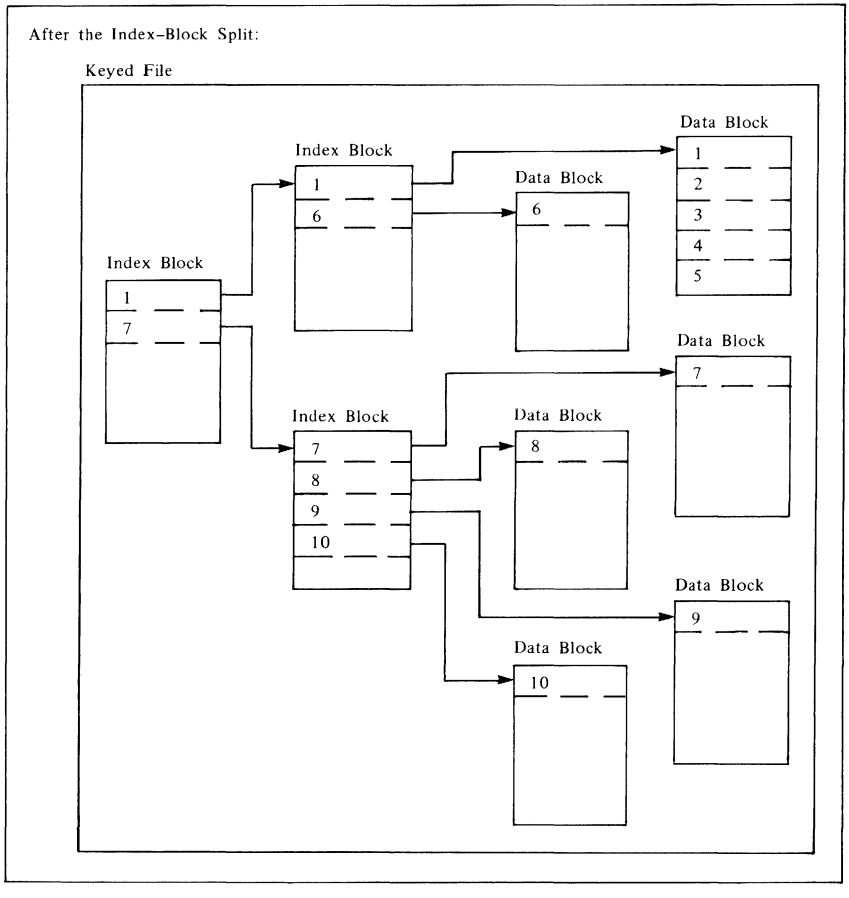


Figure I-1-3. Index-Block Split

Indexed-Sequential Primary Keys

The primary key for a keyed file is defined when the file is created. The primary-key value must be unique for each record in the file.

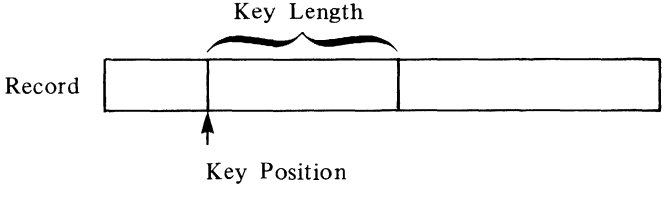
A primary-key definition requires specification of these attributes:

Embedded or nonembedded key (the default is embedded)
Key position (if the key is embedded)
Key length
Key type (the default type is uncollated_key)
Collate-table name (if the key type is collated_key)

A key is embedded if the key value is part of the data in the record. An embedded key value is returned as part of the record data when the record is read; a nonembedded key value is not.

The key position in the record must be specified if the key is embedded. The first byte position in a record is byte 0. If the key is nonembedded, you do not specify a key position.

You must specify the key length whether the key is embedded or nonembedded. It indicates the number of bytes in the key.



The key type describes the data in the key. These are the possible key types:

Integer key	The key value is a signed integer; it is sorted in numerical order.
Uncollated key	The key value is a string of characters; it is sorted byte-by-byte according to the ASCII collating sequence.
Collated key	The key value is a string of characters; it is sorted byte-by-byte according to a collating sequence that you specify.

If the key is a collated key, you must specify the collating sequence to be used to sort the key values. The collating sequence is specified by its name. NOS/VE provides several predefined collating sequences (listed in appendix E). You can also create your own collating sequence as described in appendix E.

Direct-Access File Organization

The direct-access file organization is like the indexed-sequential file organization in its use of a primary key. You define the primary key for the file when you create the file. It can be a field embedded in the record or a nonembedded value. Each primary-key value in the file must be unique; the file can contain no duplicate primary-key values.

Like an indexed-sequential file, a direct-access file can have alternate keys. An alternate key for a direct-access file is the same as an alternate key for an indexed-sequential file. Alternate keys are described later in this chapter.

Like indexed-sequential file records, you must specify the primary-key value when writing or deleting a direct-access file record. Similarly, you must specify either a primary-key value or an alternate-key value to read a direct-access file record.

Direct-access and indexed-sequential files differ in the ordering of records in the file:

- When records are read sequentially from an indexed-sequential file, the records are returned in order, sorted by primary-key value.
- When records are read sequentially from a direct-access file, the records are returned unordered.

In general, random record access is faster for the direct-access file organization than for the indexed-sequential file organization. This is because the direct-access file organization determines the location of a record directly from its primary-key value. (In indexed-sequential files, a record can be found only after a search at each index level.)

Direct-Access File Structure

The direct-access file structure is designed to locate each record directly by its primary-key value. The primary-key value directly specifies the file block containing the record.

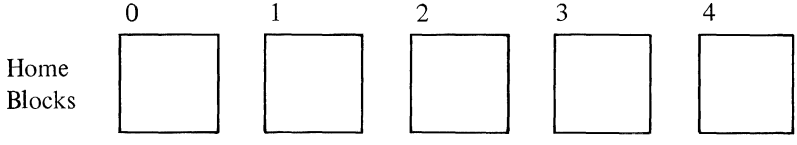
File space in a direct access file is divided into equal-size blocks. Initially, all blocks in the file are home blocks (as opposed to overflow blocks).

When a record is written to a direct-access file, its primary-key value is hashed to produce the number of the home block in which the record is written. If the home block does not contain enough empty space for the new record, the record is written to an overflow block.

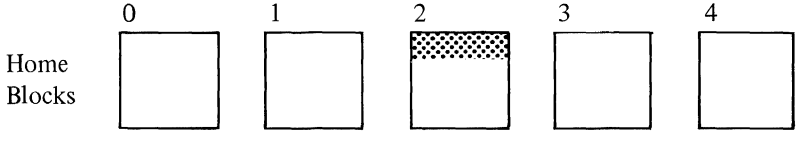
Assuming the hashing procedure produces a uniform distribution of numbers from the primary-key values in the file, the records are uniformly distributed among the home blocks of the file. Thus, each record can be found by a single search of its home block without additional searches of overflow blocks.

You specify the initial number of home blocks when you create the file. By default, a system hashing procedure is used to distribute the records among the home blocks although you can provide another hashing procedure for the file if you like.

As an illustration of a small direct-access file, suppose you define a direct access file as having five home blocks.

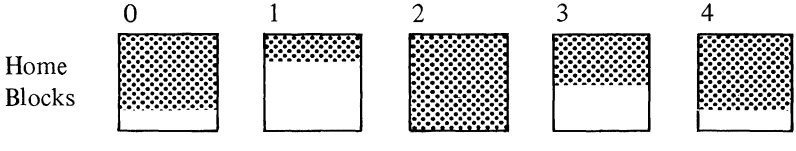


The first record written to the file has primary-key value XYZ. Assume that hashing of this primary-key value produces the block number 2. The record is then written in home block 2.

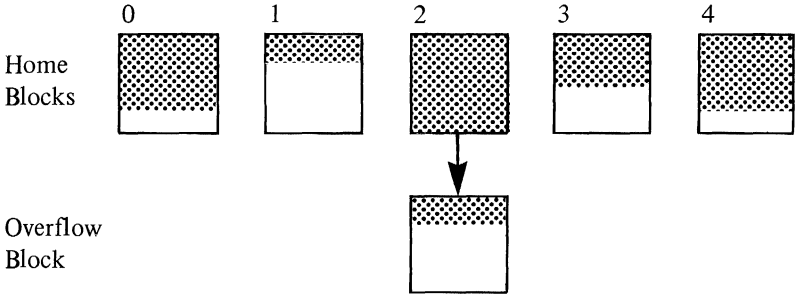


Assume you want to read the record with primary-key value XYZ. The value XYZ is hashed and, as before, produces the block number 2. The keyed-file interface searches for the record with primary-key value XYZ in home block 2. (The records in a block are ordered by primary-key value so each record can be quickly found.)

Suppose that many records have been written to the file and home block 2 has been filled.



At this point, a record is to be written with primary-key value ABC. Hashing of the value ABC produces block number 2, but there is insufficient space for the record in home block 2 so it is written in an overflow block.



Later, to read the record with primary-key value ABC, the primary-key value is hashed to produce block number 2. Home block 2 is searched for primary-key value ABC. When it is not found in the home block, the search continues in the overflow block until the record is found.

An ideal direct-access file structure has these characteristics:

- Sufficient home blocks are allocated and records are uniformly distributed among the home blocks so as to avoid overflow.
- Each block contains a limited number of records so as to minimize the search time in each block.
- The number of home blocks is not so large that the file contains excessive unused space.

These characteristics are determined by the file attribute values specified when the file is created. You must specify the initial_home_block_count and can optionally specify the max_block_length and the hashing_procedure_name attributes. (The attributes are described in chapter I-2.)

One other characteristic to be considered when selecting the number of home blocks is the loading factor. The loading factor is the percentage of block space used. To allow for less-than-uniform distribution of records in the home blocks, the loading factor should be no greater than 90%.

To illustrate, suppose the direct access file is to contain 10,000 80-byte records (80,000 bytes of record data). Using a block size of 4096 bytes, 20 home blocks would be sufficient if the hashing procedure could guarantee uniform distribution of the records in the home blocks. This would result in a loading factor of nearly 98% (80,000 divided by 81,920). However, because uniform distribution should not be expected, the number of home blocks allocated should be at least 22 (for a loading factor of 89%). (It is also recommended that the home block count be a prime number; thus, 23 would be a better home block count for the file in this example.)

Hashing Procedure

The system provides a default hashing procedure named AMP$SYSTEM_HASHING_PROCEDURE. However, if desired, you may specify your own hashing procedure that produces a uniform distribution of numbers from the primary-key values in your file.

The system executes the hashing procedure each time a record is requested by key value from the direct-access file. The hashing procedure is not stored with the file so the system must be able to load the procedure each time the direct-access file is opened.

NOTE

Any ring_attributes value is valid for the object library containing the hashing procedure. However, in a production environment, you should store the hashing procedure in a ring 4 object library. This improves performance because hashing procedures are executed as asynchronous tasks. (Usually, site personnel maintain the ring 4 object libraries.)

A hashing procedure receives a primary-key value as its input and produces an integer as its output. It must always produce the same output from a given input.

A hashing procedure is written in the CYBIL language. It must pass these parameters:

1. `primary-key value: ^cell`
   Variable in which the system passes the location of the primary-key value to be hashed.

2. `key_length: amt$key_length`
   Integer variable in which the system passes the length in bytes of the primary-key value (from 1 through 255).

3. `VAR hashed_value: integer`
   Integer variable in which the hashing procedure stores the hashed value.

4. `VAR status: ost$status`
   Standard NOS/VE status variable in which the hashing procedure stores its completion status. If the hashing procedure returns an abnormal status, the keyed-file interface issues the fatal condition aae$system_error_occurred followed by the status returned by the hashing procedure.

The system divides the value it receives from the hashing procedure by the number of home blocks and uses the remainder as the home block number. For example, if the number of blocks is 97, it divides the hashed value by 97 and uses the remainder (an integer from 0 through 96) as the home block number. A more uniform distribution of records is expected if the number of home blocks is a prime number.

Direct-Access Primary Keys

In general, the primary key of a direct-access file has the same characteristics as the primary key of an indexed-sequential file. You specify whether the primary key is embedded or nonembedded, its position (if the key is embedded), and the key length. However, a key_type attribute value specified for a direct-access file is ignored; the key_type attribute for a direct-access file is always uncollated.

Unlike an indexed-sequential file, sequential access calls to a direct-access file while the primary-key is selected do not return the file records sorted by primary-key value. The calls return records according to their physical location in the direct-access file. Records within each block are ordered according to the default ASCII collating sequence, but the blocks are not ordered by primary-key values.

Direct-access file records can be accessed in order if one or more alternate keys are defined for the file. The alternate index keeps the alternate-key values in sorted order. Sequential access calls while an alternate key is selected return records in the order provided by the alternate index.

If appropriate, you could define an alternate key for the same field as an embedded primary key. In this way, you could access direct-access file records in primary-key value order.

Alternate Keys

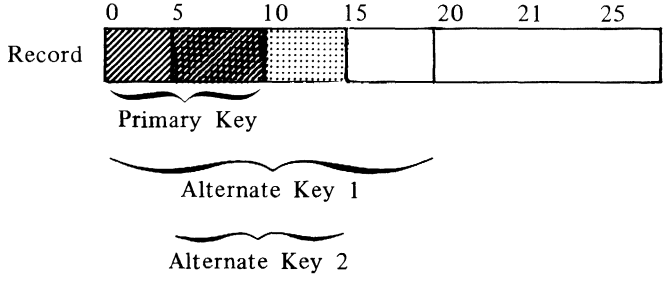
A record within a keyed file can always be accessed by its primary-key value. An alternate key provides an additional way to access records.

An alternate key defines a value in the data record by which the record can be accessed. An alternate key is defined as a field or group of fields in the record.

Although a program can use alternate keys to read records or to position a file, alternate keys cannot be used to write, replace, or delete records. The primary-key value must be used to identify a record to be written, replaced, or deleted.

Alternate-Key Characteristics

Alternate-key fields can overlap each other and an embedded primary key. For example, the primary-key field could be bytes 0 through 9 and two alternate-key fields bytes 0 through 19 and bytes 4 through 14.



Unlike a primary-key value, one alternate-key value can be associated with several records in a file. The reason is that an alternate-key value need not be unique. The same alternate-key value can occur in several records; for example, the same job title can be associated with many names, as follows:

Data Records:

```
Hanson      Computer Programmer
Jones       Computer Programmer
Smith       Computer Programmer
```

Alternate Index:

Alternate Key Value	Primary Key Values
Computer Programmer	Hanson
	Jones
	Smith

A record can contain more than one alternate-key value if the alternate key is defined as a field that repeats in the record; thus, a single record could contain several alternate-key values. For example, the license numbers of several cars owned by one person as follows:

Data Record: `R. Petty  1 LB AU  2ASM451  ELK 592`

Alternate Index:

Alternate Key Value	Primary Key Values
1 LB AU	R. Petty
2ASM451	R. Petty
ELK 592	R. Petty

The Alternate Index

The index for the primary key was described earlier in this chapter. Each alternate key defined for a file has its own index.

An alternate index contains index records, each of which associates an alternate-key value with the primary-key values of the records containing that alternate-key value. The list of primary-key values associated with an alternate-key value is the key list for that alternate-key value.

When you select an alternate key and then specify an alternate-key value, the system searches for the value in the alternate index. If it finds the alternate-key value, it uses the primary-key values in the key list for the alternate-key value to access the data records.

When one or more alternate keys are defined for a file, file updates require more time because the alternate indexes must also be updated. Alternate keys should be used only when the additional record access capability offsets the cost of increased time spent for file updates.

Alternate-Key Definition

The attributes of an alternate key are specified by its alternate-key definition.

These attributes are required to define an alternate-key:

Key name
Key position
Key length

An alternate key has a name so that it can be selected later for use. The alternate-key position and length define the alternate-key field within the record.

These optional attributes define how the alternate key is processed:

Key type
Collate table name (if the key type is collated)
Duplicate key values
Null suppression
Sparse-key control
Repeating groups
Concatenated key

The key type of an alternate key determines the order of the alternate-key values in the alternate index, and therefore, the order in which records are accessed sequentially when you use the alternate key. The key types for an alternate key are the same as the key types for the primary key as described earlier in this chapter.

If the key type is collated, you can explicitly specify a collation table for the alternate key or use, as the default, the collation table for the primary key (if the primary key type is collated).

Duplicate Key Values

By default, duplicate values for an alternate key are not allowed. However, if you want to allow duplicate key values, you can specify whether the records having the same alternate-key value are accessed ordered by primary key or in first-in-first-out order.

In a key list ordered by primary key, the primary-key values are stored in sorted order according to the primary-key type. New values are inserted into the key list so that the primary-key value order is kept.

In a key list ordered first-in-first-out, the primary-key values are stored in the key list in the order the values are added to the key list, instead of in primary-key-value order. New values are always added to the end of the key list.

For example, suppose you write three records to the file in this order:

McDarrels	Hamburgers
Burger Duke	Hamburgers
Willys	Hamburgers

The following shows the resulting key list in primary-key order and in first-in-first-out order:

	Key Lists	
Alternate Key Value	**Ordered by Primary Key**	**First In First Out**
Hamburgers	Burger Duke	McDarrels
	McDarrels	Burger Duke
	Willys	Willys

Duplicate-Key Value Error Processing

If duplicate values are not allowed and a duplicate is found in a record about to be written to the file, the record is not written to the file and a trivial error (status AAE$DUPLICATE_ALTERNATE_KEY) is returned.

A trivial error (status AAE$UNEXPECTED_DUP_ENCOUNTERED) also occurs if a duplicate value is found while a new alternate index is being created. However, the record containing the duplicate value cannot be discarded, because it is already in the file. Subsequent processing depends on whether incrementing the trivial-error count causes the count to exceed the trivial-error limit as set by the user.

- If the trivial-error limit is not exceeded, the apply operation redefines the alternate key being applied to allow duplicates, ordered by primary-key value, discards the partially built index, and builds the redefined index.
- If the trivial-error limit is reached, the apply operation returns the status condition AAE$DUPLICATE_KEY_LIMIT and removes all alternate indexes it has created. (Deleted indexes are not restored.)

In either case, a message describing the action taken is written to the $ERRORS file.

Null Suppression

By default, if an alternate-key field contains a null value, the null value is stored as the alternate-key value for the record. The null_suppression attribute allows you to exclude null values from an alternate index.

Null suppression excludes any record with a null alternate-key value from the alternate index. Null suppression can save space, access time, and update time because the index is smaller when null alternate-key values are excluded. (Null suppression does not remove the null value from the data record.)

The null value depends on the key type as follows:

Key Type	Null Value
Integer	Zero
Uncollated	Spaces
Collated	Spaces (before collation)

If null suppression is not specified, records containing a null value in the alternate-key field are indexed by the null value. The records can later be accessed by specifying the null value as the alternate-key value.

For example, suppose the spouse's name is defined as an alternate key to a membership file. Unmarried members would have a null value for the alternate-key field. Therefore, the key list for the null value lists all unmarried members. The following shows the alternate index with and without null suppression:

Without Null Suppression

Spouse's Name	Member's ID
	1626736
	8273648
Diana Simmons	4872672
Mark Ramsey	7726184
Shelly Gable	2673651

With Null Suppression

Spouse's Name	Member's ID
Diana Simmons	4872672
Mark Ramsey	2673651
Shelly Gable	7726184

Sparse-Key Control

You can use sparse-key control to create an alternate index that includes or excludes records depending on the character in a specific position in the record.

For example, suppose a student file has a one-character code indicating the student's class. To get a mailing list for juniors and seniors only, you could define an alternate index controlled by the class code.

To specify sparse-key control, you specify three values:

Value	Example
Sparse-key control position	Position of the class code in the record
Sparse-key control characters	Junior and senior class code characters
Sparse-key control effect (Indicates whether the alternate-key value should be included or excluded if the sparse-key character matches)	Included if the class code indicates a junior or senior record

Assume that the sparse-key control position is the first character after the name field and that the junior and senior class codes are 3 and 4. If the following records are copied to the file, the first three records are included in the alternate index, but not the last record.

```
Louis Skolnik     4
Gilbert Sullivan  4
Elliot Wermzer    3
Judy Manhasset    2
```

The sparse-key control position must be within the minimum record length. If you specify sparse-key control for an alternate key, the alternate-key field or fields need not be within the minimum record length.

A nonfatal (trivial) error (status AAE$SPARSE_KEY_BEYOND_EOR) is returned if both of these conditions are true for a record:

- The character at the sparse_key_control_position indicates that the record should be included in the alternate index
- The record has no alternate-key value because the record ends before the alternate-key field

When an apply or write operation detects this error, it does not include the record in the alternate index. (A write operation does write the record to the file.)

Concatenated Keys

A concatenated key is an alternate key formed from several fields, or pieces, in the record. A concatenated key can comprise up to 64 pieces.

The concatenated pieces can be noncontiguous and can be concatenated in any order. Each piece can be a different key type. All collated-key pieces use the same collation table.

The first piece you specify is the leftmost piece of the key. You specify it the same as you specify a nonconcatenated key. The pieces to be concatenated to the leftmost field are defined by individual records in the optional_attributes array. The record order in the array specifies the order of the concatenated pieces.

A concatenated key can use sparse-key control or null suppression or both. A concatenated key is considered to have a null value if the values in all fields of the key are null (before collation for collated keys).

For example, suppose you decide to define an alternate key consisting of the initials of the member's name. The first piece of the key value would be the first letter of the member's first name, the second piece would be the first letter of the member's middle name, and the third piece would be the first letter of the member's last name. Consider this data record:

0	20	40
Kennedy	John	Fitzgerald

The desired alternate key value is JFK. The concatenated-key pieces could be defined by the following CYBIL lines. (The second and third pieces are defined by records in the optional_attributes array.)

First piece (position 20, length 1):

```
AMP$CREATE_KEY_DEFINITION( fid, 'initials', 20, 1,
                           optional_attributes, status);
```

Second piece (position 40, length 1):

```
[AMC$CONCATENATED_KEY_PORTION, [40, 1, AMC$UNCOLLATED_KEY] ],
```

Third piece (position 0, length 1):

```
[AMC$CONCATENATED_KEY_PORTION, [0, 1, AMC$UNCOLLATED_KEY] ],
```

Repeating Groups

The repeating-groups attribute allows a data record to contain more than one value for the same alternate key. This allows a primary-key value to be associated with more than one alternate-key value.

To specify an alternate-key field within a repeating group:

1. Specify the first alternate-key field by its key position, key length, and key type. All subsequent alternate-key fields have the same length and type as the first.
2. Specify repeating groups for the alternate key by specifying the repeating group length: that is, the distance from the beginning of the first instance of the alternate key to the beginning of the second instance of the alternate key in the record.
3. Specify the repeating-group count: that is, how many times the alternate key field repeats in the record.

You can specify that the repeating group repeats a fixed number of times or that it repeats until the end of the record.

- If the alternate-key field repeats a fixed number of times, all alternate-key fields must be within the minimum record length.
- If the alternate-key field repeats to the end of the record, the minimum record length imposes no restriction. The system stores as many alternate-key values as the record length allows.

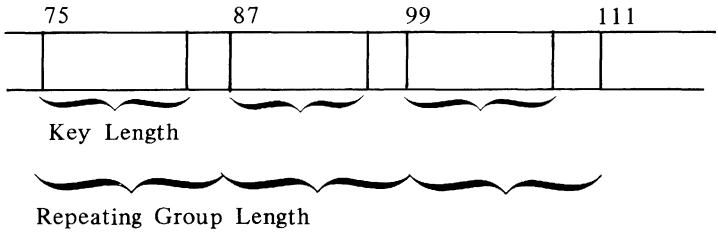
Repeating groups cannot be used with concatenated keys or when duplicate-key values are allowed and ordered first-in-first-out.

For example, suppose each record in a membership file lists the sports the member enjoys and his or her years of experience as follows (columns are counted from zero):

Field: Sports and Sports Experience

Columns: Variable number of 2-field pairs beginning at column 75 The Sports field is 10 characters followed by a 2-digit Sports Experience field

Type: ASCII characters



You could define an alternate key for the Sports values (without the Sports-Experience values) by the following CYBIL lines. (The first two lines initialize records in the optional_attributes array.)

```
{ Repeating_Group_Length=12, Repeat_to_End_of_Record=true }
[AMC$REPEATING_GROUP, [12, TRUE] ],
[AMC$DUPLICATE_KEYS, AMC$ORDERED_BY_PRIMARY_KEY],

AMP$CREATE_KEY_DEFINITION( fid, 'sports', 75, 10,
                           ^optional_attributes, status);
```

The key list for an alternate-key value would list the identification numbers of all members that enjoy that sport.

The following shows the primary keys for three records and their contents from column 75 to the end of the record:

Primary Key	Record Contents Beginning at Column 75
1662876	Volleyball02Running 03Basketball02
6166287	Bicycling 10Volleyball01
0027840	Running 15Running 15Running 15

If these were the only records in the file, the alternate index would appear as follows:

Alternate-Key Value	Primary-Key Values
Basketball	1662876
Bicycling	6166287
Running	0027840 1662876
Volleyball	1662876 6166287

Notice that the key type is the default, Uncollated_Key, and the duplicate-key values specification is Ordered_By_Primary_Key. Thus, each key list is sorted according to the default ASCII collating sequence.

Notice also, as shown by the Running key list, that each primary-key value is listed only once in a key list, regardless of the number of times the alternate-key value occurs in the record.

Nested Files

A nested file is a file structure defined within a NOS/VE file cycle. It is recognized and used by the keyed-file interface; it is not recognized or used by the NOS/VE file system.

The keyed-file interface provides nested files so as to extend the NOS/VE limit on the number of files a task can use. All nested files defined in a file share the same memory segment. This provides effective memory use when the nested files are much smaller than the segment size limit (2^{32} bytes).

The keyed-file interface creates the initial nested file (named $MAIN_FILE) when it creates the keyed file. It uses $MAIN_FILE as the default nested file; other nested files are used only when explicitly selected.

An AMP$CREATE_NESTED_FILE call can create a nested file (in addition to the default nested file $MAIN_FILE). The call defines the attributes applicable to the nested file only. These include its:

File organization

Record attributes, including its record type and its minimum and maximum record lengths

Primary-key attributes, including its key position, key length, key type, and collation table

Structural attributes applicable to the file organization

All other file attributes apply to all nested files in a keyed file. The RECORD_LIMIT attribute specifies the maximum number of records in each nested file. For more information on attributes, see Creating a Keyed File later in chapter I-2.

Each alternate-key definition applies to only one nested file. To define an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then define the alternate key. Similarly, to select an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then select the alternate key.

A task can perform operations only on the currently selected nested file. However, file position and key selection information for a nested file is not lost when another nested file is selected. For example, consider this sequence of events:

1. A task is issuing AMP$GET_NEXT calls to NESTED_FILE_1 using ALTERNATE_KEY_1
2. The task selects and uses NESTED_FILE_2.
3. The task selects NESTED_FILE_1 again. It can continue reading records sequentially from the position it had when it selected NESTED_FILE_2. The same key, ALTERNATE_KEY_1, remains selected.

The calls to manipulate nested files are described in chapter I-3. The calls are:

AMP$CREATE_NESTED_FILE
: Defines a nested file

AMP$DELETE_NESTED_FILE
: Destroys a nested file

AMP$SELECT_NESTED_FILE
: Changes the nested file currently selected

AMP$GET_NESTED_FILE_DEFINITIONS
: Returns the nested-file definitions from a keyed file

A CYBIL program demonstrating use of nested-file calls is included at the end of chapter I-2.

Using the CYBIL Keyed-File Interface I-2

Creating a Keyed File I-2-1
Setting File Attributes I-2-1
File_Organization Attribute I-2-2
Record Attributes I-2-2
Primary-Key Attributes I-2-3
File Structure Attributes I-2-4
Processing Attributes I-2-8
Writing Records I-2-10
Re-creating a Keyed File I-2-10
Using a Keyed File I-2-12
Positioning a Keyed File I-2-13
Positioning a Direct-Access File I-2-13
Positioning an Indexed-Sequential File I-2-13
Reading Records I-2-15
Sequential Access for Indexed-Sequential Files I-2-15
Sequential Access for Direct-Access Files I-2-16
Random Access I-2-17
Keyed-File Sharing I-2-18
Sharing Temporary Keyed Files I-2-19
Sharing Permanent Keyed Files I-2-19
Lock Processing I-2-21
Reasons for Locks I-2-22
Lock Intents I-2-24
Waiting for a Lock I-2-26
Lock Expiration and Clearing I-2-26
Lock Deadlock I-2-29
File Locks I-2-30
Effect of Locks on Keyed-File Calls I-2-31
Creating and Deleting Alternate Keys I-2-32
Using Alternate Keys I-2-33
Selecting an Alternate Key I-2-33
File Positioning After Alternate-Key Selection I-2-34
Reading Records After Alternate-Key Selection I-2-34
Updating an Alternate Index I-2-35
Fetching Access Information After Alternate-Key Selection I-2-36
File Position Returned I-2-37
Retrieving Alternate-Index Information I-2-38
Program Examples I-2-40
Indexed-Sequential File Creation Example I-2-41
Indexed-Sequential File Update Example I-2-45
Alternate Key Example I-2-49
Nested File Example I-2-54

Using the CYBIL Keyed-File Interface I-2

This chapter describes how CYBIL programs can create and use keyed files. A set of complete program examples is provided at the end of the chapter.

Creating a Keyed File

To create a keyed file, the following steps are required:

1. Set file attributes (AMP$FILE or AMP$OPEN calls or SET_FILE_ATTRIBUTES commands).
2. Open the file (AMP$OPEN call).
3. Optionally, write records to the file (AMP$PUT_KEY or AMP$PUT_NEXT calls).
4. Close the file (AMP$CLOSE call).

(The AMP$FILE, AMP$OPEN, and AMP$CLOSE calls are described in the CYBIL File Management manual. AMP$PUT_NEXT is described in the CYBIL Sequential and Byte Addressable Files manual. AMP$PUT_KEY is described in this manual.)

Setting File Attributes

You specify the file attributes defining the structure of the file and processing limitations for the file before opening the file for the first time. When a new file is opened, the file attributes are stored in the file; the system references the attribute values whenever the file is processed.

As described in the CYBIL File Management manual, the attributes that define the file structure cannot be changed after the file is first opened.

You should select file attribute values carefully. Selecting suitable values for file attributes helps ensure that the file economizes both space and the time needed for record retrievals.

NOTE

Most attributes have a default value. However, the default value is sometimes inappropriate for keyed files. Therefore, it is recommended that you explicitly specify a value for all relevant keyed-file attributes.

File_Organization Attribute

To create a keyed file, you specify a keyed-file organization as the file_organization attribute. Currently, the keyed-file organizations are indexed-sequential and direct-access.

To specify indexed-sequential file organization, you initialize an attribute record as follows:

```
[AMC$FILE_ORGANIZATION, AMC$INDEXED_SEQUENTIAL]
```

To specify direct-access file organization, you initialize an attribute record as follows:

```
[AMC$FILE_ORGANIZATION, AMC$DIRECT_ACCESS]
```

The other keyed-file attributes define record attributes, primary key attributes, file structure attributes, and processing attributes.

Record Attributes

These attributes describe the data records to be written to the keyed file.

NOTE

The record attributes are all preserved attributes, that is, the attribute value is stored with the file when the file is first opened and cannot be changed thereafter.

The following lists the CYBIL attribute identifier (AMC$xxx) followed by the valid attribute values:

AMC$RECORD_TYPE

Record type: AMC$FIXED, AMC$VARIABLE, or AMC$UNDEFINED. The default is AMC$UNDEFINED.

AMC$MAX_RECORD_LENGTH

Maximum number of bytes in a data record (from 1 through 65497). You must specify a value for this attribute when defining a keyed file.

AMC$MIN_RECORD_LENGTH

Minimum number of bytes in a data record (from 0 through 65497).

If the AMC$RECORD_TYPE value is AMC$ANSI_FIXED, the default minimum record length is the AMC$MAXIMUM_RECORD_LENGTH value. If the AMC$RECORD_TYPE value is AMC$UNDEFINED or AMC$VARIABLE and the key is embedded, the default is the sum of the AMC$KEY_POSITION and AMC$KEY_LENGTH values. Otherwise, the default is 1.

For variable-length records, explicit specification of this attribute is recommended; the minimum record length must include:

- The primary-key field
- Any alternate-key fields (or corresponding sparse-key control characters)
- All alternate-key fields for an alternate key defined as a field in a repeating group which repeats a fixed number of times

Primary-Key Attributes

These attributes define the primary key of the new file. See Primary Keys earlier in this chapter for more information on primary keys.

NOTE

The primary-key attributes are all preserved attributes. That is, the attribute value is stored with the file when the file is first opened and cannot be changed thereafter.

The following lists the CYBIL attribute identifier (AMC$xxx) followed by the valid attribute values:

AMC$EMBEDDED_KEY

Boolean value indicating whether the primary key is part of the record data (embedded) or separate from the record data (nonembedded). The default is TRUE (embedded keys).

AMC$KEY_LENGTH

Integer specifying the primary-key length in bytes. This attribute has no default value; it must be defined before the file is first opened.

AMC$KEY_POSITION

Position of the leftmost byte in the primary key (specified only if the key is embedded). The byte positions in a record are numbered from the left, beginning with 0. The default is 0.

AMC$KEY_TYPE

Primary key type: AMC$UNCOLLATED_KEY, AMC$INTEGER_KEY, or AMC$COLLATED_KEY. The default is AMC$UNCOLLATED_KEY.

For direct-access files, any value specified for the key_type attribute is ignored. The key_type for a direct-access file is always uncollated.

AMC$COLLATE_TABLE_NAME

Name of the collating sequence by which collated keys are ordered (required if the key_type is collated).

The name can be the name of a NOS/VE predefined collating sequence or, for a user-defined collating sequence, the name of an entry point in an object library. See appendix D for more information.

File Structure Attributes

These attributes affect the internal file structure. Keyed-file structure is described in chapter I-2.

The first group of attributes applies to all keyed-file organizations; the groups that follow each apply to one keyed-file organization only.

NOTE

The file structure attributes are all preserved attributes. That is, the attribute value is stored with the file when the file is first opened and (except for record_limit) cannot be changed thereafter.

Common File Structure Attributes

The following lists the file structure attributes common to all keyed-file organizations. It lists the CYBIL attribute identifier (AMC$xxx) followed by the valid attribute values:

AMC$RECORD_LIMIT

Maximum number of data records allowed in each nested file in the file (integer from 1 through 2^{42}-1).

The record_limit attribute value can be changed by the CHANGE_FILE_ATTRIBUTES command even after the file has been opened. For more information, see the SCL System Interface Usage manual.

AMC$MAX_BLOCK_LENGTH

Number of bytes in each block (integer from 1 through 16777215 [2^{24}-1]).

If the value is less than the maximum record length, the system increases it to that value. Then, if the value is not a power of 2 between 2048 and 65536, it changes the value as follows:

- If the value is less than 2048, it is increased to 2048 (the minimum allocation unit).
- If the value is between 2048 and 65536, but not a power of 2, it is increased to the next power of 2 (4096, 8192, 16384, 32768, or 65536).
- If the value is greater than 65536, it is decreased to 65536.

NOTE

If the file will be shared by more than one concurrent instance of open and forced-writing will be used (the FORCED_WRITE attribute is either AMC$FORCED or AMC$FORCED_IF_STRUCTURE_CHANGE), its block size should be a multiple of a system page size. This ensures that more than one instance of open is not updating blocks in the same page; otherwise, a forced-write operation could write a page to mass storage that contains partially-altered blocks. (A warning message is issued if this situation exists.)

It is recommended that you do not specify the block length as the AMC$MAX_BLOCK_LENGTH attribute, but rather allow the system to calculate the block length using values specified by the following attributes.

Block Length Guideline Attributes

NOTE

The following attributes do not set limits; their values are used only as guidelines for determining the block length when the file is created.

AMC$AVERAGE_RECORD_LENGTH

Estimated median record length, in bytes, of the data records to be stored in the file. (The length should not include a nonembedded key.)

If you omit this parameter, the system uses the arithmetic mean between the maximum and minimum record lengths in its calculation of the block size.

AMC$ESTIMATED_RECORD_COUNT

Estimated number of data records to be stored in the file. If you do not define this attribute, the system uses in its calculation of the block size either the AMC$RECORD_LIMIT value, or if that attribute is not defined, the value 100,000.

AMC$INDEX_LEVELS

Target number of index levels for the file (0 through 15). The default value is 2.

This attribute applies only to indexed-sequential files.

AMC$RECORDS_PER_BLOCK

Estimated number of data records to be stored in each data block. If you do not define this attribute, the system uses the value 2 in its calculation of the block size.

Indexed-Sequential Structure Attributes

The following structure attributes apply only to indexed-sequential files.

AMC$DATA_PADDING

Percentage of data-block space left empty when a block is created (integer). The default is 0% (no padding). The percentage must allow for storage of at least one maximum-length record per block.

AMC$INDEX_PADDING

Percentage of index-block space left empty when a block is created (integer). The default is 0% (no padding). The percentage must allow for storage of at least three index records per block. (The index record length is the key length plus 4.)

Direct-Access Structure Attributes

The following structure attributes apply only to direct-access files.

AMC$INITIAL_HOME_BLOCK_COUNT

Number of home blocks in the file (1 through $2^{31}-1$ [the segment size limit divided by the minimum allocation unit]).

NOTE

Specification of this attribute is required when creating a direct-access file.

For best results, the number should be a prime number. You should consider the expected number of records in the file and the block size when selecting the number of home blocks. For more information, see the discussion under Direct-Access File Structure in chapter I-1.

AMC$HASHING_PROCEDURE_NAME

Pointer to a record identifying the hashing procedure to be executed with this file (^amt$hashing_procedure_name). The record has these fields:

NAME	Entry point name of the hashing procedure (pmt$program_name). All letters in the name must be specified as uppercase.
OBJECT_LIBRARY	File path to the object library containing the hashing procedure (amt$path_name, 256-character string). This feature is currently unimplemented; specify OSC$NULL_NAME as the field value.

The default hashing procedure is the one provided by the system, entry point AMP$SYSTEM_HASHING_PROCEDURE.

If a hashing procedure other than the default is specified, it must be a procedure declared with the XDCL attribute within the global library set of the job or defined within the task. The hashing procedure must be available whenever the file is used; otherwise, AMP$OPEN returns the condition aae$cant_load_hash_routine.

Processing Attributes

These attributes set keyed-file processing options.

NOTE

The forced_write and lock_expiration_time attributes are preserved attributes, but their values can be changed by the CHANGE_FILE_ATTRIBUTES command. For more information, see the SCL System Interface Usage manual.

The error_limit and message_control attributes are temporary attributes; their values can be changed each time the file is opened.

AMC$ERROR_LIMIT

Maximum number of trivial (nonfatal) errors that can occur before the trivial errors cause a fatal error. The default value is 0, meaning no limit.

AMC$FORCED_WRITE

Identifier indicating when the system copies modified blocks to mass storage.

AMC$FORCED	Write modified blocks immediately.
AMC$UNFORCED	Allow modified blocks to remain in memory until the next flush or close request.
AMC$FORCED_IF_STRUCTURE_CHANGE	Write modified blocks immediately if the change affects more than one block.

The default value is AMC$FORCED_IF_STRUCTURE_CHANGE.

AMC$LOCK_EXPIRATION_TIME

Number of milliseconds between the time a lock is granted and the time that it could expire (integer from 0 through 604,800,000 [1 week]). (Preserved attribute.)

The default value is 0. When the lock expiration time is 0, locks do not expire.

This attribute value can be changed by a CHANGE_FILE_ATTRIBUTES command.

AMC$MESSAGE_CONTROL

Indicates the additional information written to the $ERRORS file besides fatal error messages. The attribute value is specified as a set in the set identifier AMTMESSAGE_CONTROL[ ].

AMC$TRIVIAL_ERRORS	Nonfatal-error messages
AMC$MESSAGES	Informative messages
AMC$STATISTICS	Statistical messages
Null set	Suppress nonfatal-error, informative, and statistical messages.

The default value is the null set.

Writing Records

Records can be written to a keyed file opened with at least append access. (If alternate keys are defined for the file, it must be opened with modify, append, and shorten access.)

You can write records to a new keyed file using either AMP$PUT_KEY or AMP$PUT_NEXT calls. Use of AMP$PUT_KEY calls is recommended for writing keyed files. AMP$PUT_NEXT should be used only if a common interface for writing records, regardless of file organization, is required.

NOTE

An AMP$PUT_NEXT call cannot specify a key value. When the keyed file has a nonembedded primary key, AMP$PUT_NEXT takes the key value from the beginning of the working storage area. It stores the first key_length bytes as the nonembedded primary-key value and the rest of the data as the record.

In general, pre-sorting records to be written to an indexed-sequential file can result in a smaller file and less time required for writing the records. Your program can use NOS/VE Sort/Merge to sort records as described in part II of this manual.

For an indexed-sequential file with an embedded primary key, you could use NOS/VE Sort/Merge calls to write the original set of records to the file. (NOS/VE Sort/Merge calls are described in part II of this manual.) The Sort/Merge specification must define the primary-key field as the major sort key.

Re-creating a Keyed File

As described earlier, the initial keyed-file structure is created when the file is first opened using the file structure attribute values defined for the file. As records are added, replaced, and deleted in the file, the file structure may become inefficient. When this becomes evident, you should re-create the file to improve the efficiency of its structure.

The evidence of an inefficient file structure differs depending on the keyed-file organization.

As described at the beginning of this chapter, record access in an indexed-sequential file is through a hierarchy of index blocks. Each additional index level in the hierarchy requires an additional index block search for each data record access. Performance is usually best when no more than two index levels exist.

You can fetch the current number of index levels in an indexed-sequential file as the levels_of_indexing file access information item using the AMP$FETCH_ACCESS_INFORMATION call. (The AMP$FETCH_ACCESS_INFORMATION call is described in the CYBIL File Management manual.)

An inefficient direct-access file structure is indicated by an excessive number of overflow blocks and overflow records. The overflow_block_count and overflow_record_count for the file are included in the list of structural properties provided by the DISPLAY_KEYED_FILE_PROPERTIES command.

To re-create a keyed file, you first define the structural attributes for the re-created file and then copy the old file to the newly defined file. You can copy the file by any of these means:

- Executing the SCL command COPY_KEYED_FILE (described in the SCL Advanced File Management manual)
- Calling AMP$COPY_FILE as described in the CYBIL Sequential and Byte Addressable Files manual
- Using the File Management Utility (FMU) as described in the SCL Advanced File Management manual. (Unlike the preceding two methods, FMU can reformat and selectively copy records while re-creating the file.)

The COPY_KEYED_FILE command can apply the alternate-key definitions from the old file to the new file. AMP$COPY_FILE and FMU do not apply alternate-key definitions.

If you did not use COPY_KEYED_FILE to re-create the file, you can re-create alternate keys by this method:

1. Save the alternate-key definitions from the old keyed file on a file. To get the alternate key definitions used by the file, call AMP$GET_KEY_DEFINITIONS.
2. Use the saved definitions to redefine the alternate keys on the new file. To do so, open the new file, call AMP$CREATE_KEY_DEFINITION to specify each alternate-key definition, and then apply the definitions with an AMP$APPLY_KEY_DEFINITIONS call.

Using a Keyed File

To process an existing keyed file, a CYBIL program performs these steps:

1. Specifies temporary attribute values to be used by this instance of open and preserved attribute values to be verified against the attribute values stored with the file (AMP$FILE and AMP$OPEN).
2. Opens the keyed file for record access (AMP$OPEN).
3. Performs the intended file operations.
4. Closes the file (AMP$CLOSE).

The following file operations can be performed on an existing keyed file (assuming the file has been opened with the required access modes):

- Position the file (AMPGET_KEY, AMPREWIND, AMP$SKIP, and AMP$START).
- Read records randomly by key value (AMP$GET_KEY).
- Read records sequentially by position (AMP$GET_NEXT_KEY and AMP$GET_NEXT).
- Write records (AMPPUT_KEY, AMPPUT_NEXT, and AMP$PUTREP).
- Delete records (AMP$DELETE_KEY).
- Replace existing records (AMP$REPLACE_KEY and AMP$PUTREP).
- Lock key values (AMP$LOCK_KEY, AMP$GET_LOCK_KEYED_RECORD, AMP$GET_LOCK_NEXT_KEYED_RECORD, and AMP$LOCK_FILE).
- Unlock key values (AMP$UNLOCK_KEY and AMP$UNLOCK_FILE).
- Define, delete, and select nested files (AMP$CREATE_NESTED_FILE, AMP$DELETE_NESTED_FILE, AMP$GET_NESTED_FILE_DEFINITIONS, and AMP$SELECT_NESTED_FILE).
- Define, delete, and select alternate keys as described later in this chapter.

Depending on the value of the forced_write attribute, the system might not write modified blocks to mass storage immediately after the modification. You can call AMP$FLUSH any time after the file is opened to write the part of the file in memory to mass storage. Execution of the AMP$FLUSH call does not change the position of the file.

Positioning a Keyed File

To position a keyed file, a program must open the file for at least read access. In general, a program positions a file so that it can later read records sequentially.

For information on positioning a file by alternate-key values, refer to Using Alternate Keys later in this chapter.

As described later under Reading Records, the sequential access capabilities differ for indexed-sequential and direct-access files. This results in differences in the positioning calls available for each organization.

Positioning a Direct-Access File

While an alternate key is selected, the same positioning calls are valid for a direct-access file as for an indexed-sequential file. However, while the primary key is selected, the only valid positioning call is AMP$REWIND. AMP$REWIND positions a direct-access file at the beginning of its first home block.

While the primary key is selected, an AMP$SKIP call specifying a direct-access file returns the nonfatal condition aae$no_skip_in_da. An AMP$START call for a direct-access file with its primary key selected returns the condition aae$no_da_or_sk_start.

Positioning an Indexed-Sequential File

The following positioning calls are available for indexed-sequential files:

- AMP$GET_KEY: Returns to the working storage area the record whose key value matches the key value specified on the call and positions the file at the end of the returned record.
- AMP$REWIND: Positions a file to read the record with the lowest key value.
- AMP$SKIP: Positions a file forward or backward.
- AMP$START: Positions a file to read the record whose key value matches the key value specified on the call.

Positioning an Indexed-Sequential File by Major Key

The AMP$START, AMP$GET_KEY, and AMP$GET_LOCK_KEYED_RECORD calls have a major_key_length parameter. This parameter allows a call to position an indexed-sequential file according to a major-key value.

A major key consists of one or more of the leftmost bytes of a key. The major_key_length parameter specifies the number of bytes to use as the major key. A major key search compares only the number of bytes in the major key.

For example, suppose the key value at the specified key_location is ABCDEF and the major_key_length parameter value is 2. The major-key value, therefore, is the leftmost two bytes, characters AB. The major key search compares the characters AB with the leftmost two bytes of the searched keys. It positions the file at the first record whose key begins with AB or greater.

As a second example, suppose the key value is the hexadecimal integer FF145 and the major key length value is 3. The major key used is the leftmost three bytes containing the value FF1, so the file is positioned at the first record whose key begins with FF1 or greater.

If the major_key_length parameter is zero or equal to key_length, the entire key is used to position the file.

The major_key_length parameter is ignored on direct-access file calls.

Positioning an Indexed-Sequential File by Key Relation

The AMPGET_KEY, AMPGET_LOCK_KEYED_RECORD, and AMP$START calls have a key_relation parameter. This parameter allows a call to position an indexed-sequential file even if the specified key value does not exist in the file.

The key_relation parameter specifies the relation to be satisfied between the specified key value and the key value of the record at which the file is positioned. The relation can be equal, greater than or equal, or greater than.

For example, suppose the specified key value is ABC.

- If the specified key_relation is equal, the call must find a record whose key value matches ABC. If such a record is not found, the call returns an abnormal completion status.
- If the specified key_relation is greater than or equal to, the first key value found that is greater than or equal to ABC satisfies the relation. If the relation cannot be satisfied, the file is left positioned at its end-of-information.

- If the specified key_relation is greater than, the first key value found that is greater than ABC satisfies the relation. If no key value is greater than ABC, the file is left positioned at its end-of-information.

The key_relation parameter is ignored for direct-access file calls.

Reading Records

For records to be read from a keyed file, the file must be open for at least read access. However, it is recommended that the file be opened for both read and modify access. Modify access allows access statistics to be updated without allowing any record in the file to be altered.

A read operation transfers a record from the file to the specified working storage area. The number of bytes in the record is returned in the record_length parameter.

You cannot call AMP$GET_PARTIAL to read a keyed-file record. However, a partial read of a record is performed when the record is longer than the working storage area specified on the get call. The get call reads data until the working storage area is filled and then returns a nonfatal error (AAE$RECORD_LARGER_THAN_WSA). The get call leaves the keyed file positioned at the end of the record; thus, the next read request cannot begin where the partial read ended.

You can read records either sequentially by position or randomly by key value. A sequential read returns the next logical record in the file. A random read returns the record identified by the specified key value.

Sequential Access for Indexed-Sequential Files

Records can be read sequentially from an indexed-sequential file using AMP$GET_NEXT_KEY or AMP$GET_NEXT calls. Use of AMP$GET_NEXT_KEY calls is recommended for reading indexed-sequential files. You should use AMP$GET_NEXT only if a common interface for writing records, regardless of file organization, is required.

AMP$GET_NEXT_KEY returns the key value of each record in the location specified by the key_location parameter. The task can check the file_position value returned to determine when to stop reading records.

You can also read a contiguous group of records residing anywhere in the file by combining random access and sequential access. This is accomplished by issuing an AMP$GET_KEY to read the first record in the contiguous group, and, then, issuing AMP$GET_NEXT_KEY calls (or AMP$GET_NEXT) to read the following records sequentially.

Sequential Access for Direct-Access Files

Records are not stored in sorted order by primary-key value in direct-access files as they are in indexed-sequential files. Thus, sequential access is appropriate only:

- When an alternate key is selected
- When a primary key is selected and all records in the file are to be read

A sequential pass through a direct-access file is valid only when no update operation intervenes. An intervening update operation could cause the sequential pass to miss records. (Sequential access to a direct-access file is done by physical position in the file; an update operation could change the record locations.)

To provide effective sequential access, the keyed-file interface imposes these restrictions on sequential access to direct-access files:

- When the primary key is selected, AMP$GET_LOCK_NEXT_KEY, AMP$GET_NEXT_KEY and AMP$GET_NEXT calls are valid only when the direct-access file has been attached for exclusive access (no share modes allowed).

  When the primary key is selected and the file attachment allows sharing, a sequential get call returns the condition aae$cant_da_getn_if_shared.

- When the primary key is selected, a program cannot intermix sequential access calls and update operations. (The only update operation allowed is the replacement of a record with another record of the same length.)

  When the primary key is selected and an update operation has been performed, the program must rewind the file before beginning a sequential pass of the direct-access file. Otherwise, a sequential get call returns the condition aae$cant_da_getn_after_put.

You can intermix sequential access (get_next) calls and AMP$GET_KEY calls. An AMP$GET_KEY call does not change the file position used by get_next calls.

Random Access

Records are read randomly by key value using the AMP$GET_KEY call. To retrieve a single record from the keyed file, you specify a key value, and the system returns to the working storage area the record with the matching key value, if it exists.

For indexed-sequential files, the major_key_length parameter allows the AMP$GET_KEY call to read the first record with the specified major key. The key_relation parameter allows AMP$GET_KEY to specify the relation between the key value of the record to be read and the specified key value. The relation could be equal, greater than, or greater than or equal.

The major_key_length and key_relation parameters are ignored on direct-access file calls.

Keyed-File Sharing

A NOS/VE keyed file can be accessed with or without potential sharing of the file. A keyed file is shared when multiple concurrent instances of open of the file exist.

The potential for sharing determines whether NOS/VE must safeguard the keyed-file structure for multiple users:

- While a keyed file could be shared, NOS/VE performs internal locking operations to maintain the integrity of the file structure.
- While a keyed file cannot be shared, the overhead required to maintain file integrity is not needed, resulting in better file access performance.

File access is controlled by the set of access modes in effect for the file. File sharing is controlled by the set of share modes in effect. The use of access modes and share modes for NOS/VE files in general is described in the SCL System Interface and CYBIL File Management manuals; access mode and share mode use for keyed files is described here.

To see the access modes and share modes currently in effect for a file, enter this SCL command (specifying the file name or file reference):

```
Display_File_Attributes, File=file, ..
  Display_Options=(Access_Modes, Global_Share_Modes)
```

The Access_Modes set is the set of access modes currently in effect. It is contained in the Global_Access_Modes set (the set of all available access modes as determined when the file is created or attached). When the file is created or attached, the Access_Modes and Global_Access_Modes values sets are the same. However, the Access_Modes set can be restricted to a subset of the Global_Access_Modes by a SET_FILE_ATTRIBUTES command or AMP$FILE or AMP$OPEN call. Keyed-file sharing is affected only by the Access_Modes set; the Global_Access_Modes set only indicates the possible values of the Access_Modes set.

The Global_Share_Modes set is the set of share modes currently in effect. It is determined when the file is created or attached; you cannot change the Global_Share_Modes using SET_FILE_ATTRIBUTES commands or AMP$FILE or AMP$OPEN calls.

AMP$GET_FILE_ATTRIBUTES and AMP$FETCH calls in a CYBIL program can fetch the Access_Modes and Global_Share_Modes sets.

Sharing Temporary Keyed Files

You can specify the Access_Modes currently in effect for a permanent or temporary keyed file. However, because you can specify Share_Modes only when attaching the file, you cannot specify Share_Modes for a temporary file. The Global_Share_Modes value for a temporary file is always none.

Thus, a temporary keyed file cannot be shared. It can be opened consecutively within a job, but it cannot be opened concurrently, that is, it cannot have multiple instances of open.

To illustrate how tasks can open a temporary keyed file, suppose task X creates and opens a temporary keyed file. Task X cannot open the file again until it closes the existing instance of open. After task X closes the file, task X or task Y can then open the file. Also, if task X opens the file and then initiates task Y, task Y cannot open the file until task X closes the file.

Sharing Permanent Keyed Files

For a permanent keyed file, the Share_Modes can be explicitly specified when the file is attached; otherwise, the default set is used. NOS/VE provides two default Global_Share_Mode values as follows:

1. When the Access_Modes include any of the write modes (append, modify, or shorten), the default Global_Share_Mode value is none. Thus, by default, NOS/VE allows no sharing while the file could be changed.

   For example:

   ```
   /attach_file, $user.keyed_file, access_mode=write
   /display_file_attributes, keyed_file, ..
   ../display_options=(access_modes, global_share_modes)
   Access_Mode                : (shorten, append, modify)
   Global_Share_Mode          : none
   ```

2. When the Access_Modes do not include any of the write modes (append, modify, or shorten), the default Global_Share_Mode value is read and execute. Thus, by default, the file cannot be changed.

   For example:

   ```
   /attach_file, $user.keyed_file, access_mode=read
   /display_file_attributes, keyed_file, ..
   ../display_options=(access_modes, global_share_modes)
   Access_Mode                : (read)
   Global_Share_Mode          : (read, execute)
   ```

In the first situation, no locking is needed because no sharing is allowed. In the second situation, no locking is needed because the data cannot change. When no locking is needed, no setting of locks or checking for locks is done and performance improves.

NOTE

For best performance when using a keyed file, check that the share modes allowed are no more than those required. If possible, allow no sharing of the file.

In general, when the file can be shared (the Global_Share_Modes value is not none) and either the Access_Modes or the Global_Share_Modes include shorten or append access, locking is needed. The following examples show two situations in which locking is not needed and a third situation in which it is needed.

1. When reading a keyed file, it is recommended that you request modify access so that read statistics can be recorded in the file. Because modify is one of the write access modes, no other instances of open can access the file while you read it (if you do not explicitly specify Share_Modes). For example:

```
/attach_file, $user.keyed_file, access_modes=(read, modify)
/display_file_attributes, keyed_file, ..
../display_options=(access_modes, global_share_modes)
Access_Mode                : (read, modify)
Global_Share_Mode          : none
```

   In this case, because no sharing is allowed, no locking is performed and performance is at its best.

2. Next, to allow other users to read the keyed file and maintain accurate read statistics, you explicitly specify the Share_Modes as read and modify:

```
/attach_file, $user.keyed_file, access_modes=(read, modify) ..
../share_modes=(read, modify)
/display_file_attributes, keyed_file, ..
../display_options=(access_modes, global_share_modes)
Access_Mode                : (read, modify)
Global_Share_Mode          : (read, modify)
```

   In this case, sharing is allowed, but the file data cannot be changed. So again, no locking is performed and performance is at its best.

3. Suppose that the permit applicable to the attach allows all access modes to the file, but requires that shorten and append share modes be allowed. You choose to request all access modes and allow all share modes:

```
/attach_file, .xyz.keyed_file, access_modes=all, share_modes=all
/display_file_attributes, keyed_file, ..
../display_options=(access_modes, global_share_modes)
Access_Mode                : (read, shorten, append, modify)
Global_Share_Mode          : (read, shorten, append, modify)
```

   In this situation, other instances of attach, as well as this one, can write, replace, and delete records. Because of the potential for file sharing, NOS/VE uses internal locks as needed to maintain the integrity of the file structure. A program using the file in a shared situation such as this may choose to use locks to disallow changes to data it is currently using; it must lock the primary-key value of any record it deletes or replaces.

The reasons for using locks and the means of doing so are described in detail in the following pages.

Lock Processing

Keyed-file sharing can be coordinated through the use of locks. A lock is a mechanism by which a task can restrict use of a keyed file or individual primary-key values in keyed files. The lock is owned by a particular instance of open for the file. The part of the NOS/VE system software that manages locks is called the lock manager.

In general, lock processing follows this pattern:

1. The lock manager receives a request for a lock on a file or primary-key value.

2. The lock manager determines whether the lock can be granted.

   a. If no conflicting lock exists, the lock manager grants the lock and notifies the requesting task.

   b. If a conflicting lock exists, the lock manager checks if the request specified waiting.

      i. If the request specified no waiting, the lock manager notifies the task requesting the lock that the record or file is currently locked.

      ii. If the request specified waiting, the task is suspended until either:

         - The lock is available (assuming no potential deadlock as described later under Lock Deadlock), or

         - The timeout period elapses (default value, 60 seconds).

The lock manager also processes requests to clear locks and keeps track of locks that have expired (as described later under Lock Expiration and Clearing).

NOTE

In general, when the discussion of locks in this manual describes two or more tasks requesting locks, the two or more tasks could actually be the same task with two or more instances of open of the same file. This is because a lock belongs to a particular instance of open and one task could be requesting locks for more than one instance of open.

Lock use is recommended for effective sharing of a keyed file. In fact, when more than one instance of open exists for a keyed file, NOS/VE requires that a task lock the record before it can replace or delete the record.

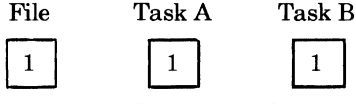
Lock use ensures that:

- Requests are processed in the sequence in which requests are issued.
- The operation is performed on the most up-to-date version.

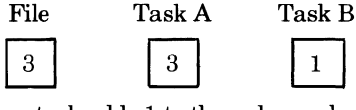
Reasons for Locks

To illustrate the need for locks, the following sequence of events describes two tasks using the same file without locks.

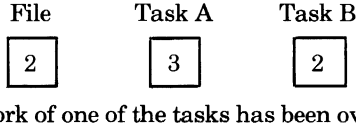
1. Two tasks both read the same record containing the value 1.



2. One task adds 2 to the value and replaces the record, containing the value 3, in the file.



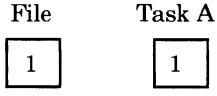
3. The other task adds 1 to the value and replaces the record, containing the value 2, in the file.



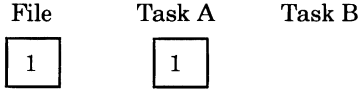
The work of one of the tasks has been overwritten.

In contrast, consider the following sequence of events describing two tasks using the same file with locks.

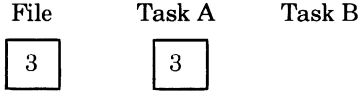
1. A task locks and reads a record.



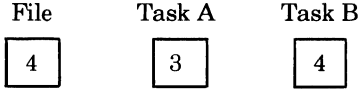
2. A second task attempts to lock and read the record but cannot because the record is already locked. It waits until the record is unlocked.



3. The first task adds 2 to the value, and replaces the record containing the value 3, in the file. It then unlocks the record.



4. The second task can now lock and read the record. It adds 1 to the value, and replaces the record, containing the value 4, in the file.



Lock Intents

Each lock has a lock intent. The lock intent indicates why the task is requesting the lock.

When more than one instance of open exists for a keyed file, only the owner of an Exclusive_Access or Preserve_Access_and_Content lock on the record (or the file) can replace or delete the record. However, the replace or delete operation does not take place until no unexpired Preserve_Content locks exist for the record.

The following paragraphs describe the lock intents for record locks. (Lock intents for file locks are described later under File Locks.)

Exclusive_Access

- Used when the task intends to issue write or delete requests for the locked record.
- Denies all requests by other tasks to read, write, update, or delete the record or lock its key value.
- Allow requests by other tasks that position the file or perform operations only on alternate indexes.

Preserve_Access_and_Content

- Used when the task might issue write or delete requests for the locked record. Only one Preserve_Access_and_Content lock is allowed at a time for a record.
- Allows positioning and read requests by other tasks, but denies their write, replace, and delete requests.
- Allows Preserve_Content lock requests by other tasks, but denies their requests for Exclusive_Access and Preserve_Access_and_Content locks on the record.
- The owner of the Preserve_Access_and_Content lock can request a write, replace, or delete operation, but:
  - The write, replace, or delete operation does not begin until the conditions for an Exclusive_Access lock are met:
    - All read operations in progress for the record have completed.
    - All Preserve_Content locks for the record have expired or been cleared.
  - No read operations for the record can begin until the write, replace, or delete operation completes.

Preserve_Content

- Used when the task does not intend to issue write, replace, or delete requests for the locked record.
- Allows positioning and read requests by other tasks, but denies their write, replace, and delete requests.
- Allows Preserve_Content and Preserve_Access_and_Content locks by other tasks, but denies their Exclusive_Access lock requests.

Multiple Preserve_Content locks are allowed at a time, but only one Preserve_Access_and_Content lock. Thus, multiple tasks can be reading the record, but only one task can be waiting to write, replace, or delete the record.

Switching Lock Intents

The owner of a lock on a record can request another lock on the record with the same lock intent without an intervening unlock request.

The owner of an Exclusive_Access or Preserve_Access_and_Content lock can also switch the lock intent to Exclusive_Access or Preserve_Access_and_Content without an intervening unlock request.

A request to change the lock intent from Preserve_Access_and_Content to Exclusive_Access is not performed until any Preserve_Content locks on the record or the file are no longer effective.

A lock request that renews an existing lock restarts the expiration time for the lock.

This table summarizes the lock intent switching that is valid without an intervening unlock request.

From \ To	Exclusive_Access	Preserve_Access_and_Content	Preserve_Content
Exclusive_Access	Valid	Valid	Invalid
Preserve_Access_and_Content	Valid	Valid	Invalid
Preserve_Content	Invalid	Invalid	Valid

Waiting for a Lock

On a call that requests a lock, you specify whether the call should wait if the lock is unavailable. If you specify that the call should wait, it waits until the lock is available or a lock timeout period has passed. When the time period has passed, the call terminates with the condition aae$key_timeout.

The default timeout period is 60 seconds. However, each task can specify how long it waits for a lock by defining and initializing an SCL integer variable.

The timeout variable is named AAV$RESOLVE_TIME_LIMIT. You assign the variable the new waiting period in seconds (from 1 through 604,800,000 [1 week]).

For example, the following call executes the SCL command CREATE_VARIABLE to create the AAV$RESOLVE_TIME_LIMIT variable and assign it the value 45.

```
clp$scan_command_line('create_variable, AAV$RESOLVE_TIME_LIMIT, CAT
  kind=integer, value=45, scope=local', status);
```

(The CLP$SCAN_COMMAND_LINE call is described in the CYBIL System Interface manual.)

Lock Expiration and Clearing

An expired lock and a cleared lock are not the same:

- A cleared lock no longer exists; the lock manager has discarded it.
- An expired lock exists, but is no longer effective in preventing access by other tasks. However, an expired lock prevents file access by its owner (except to fetch or store attributes or access information). This is done so that the owner of the lock is notified of its expiration.

A lock is cleared when one of these events occurs:

- The task with the lock issues an unlock request for the lock.
- The task closes the instance of open to which the lock belongs.
- The request for the record lock specified automatic unlock, and the task issues any request for the instance of open (other than a call to fetch or store attributes or fetch access information).

In general, the automatic unlock occurs when the request is issued. The exception is for an update request for the locked record for which the lock is kept until the update operation completes.

For example, if a task issues a lock on record 1 and then issues a request to replace record 1, the lock manager automatically clears the lock on record 1 after the replace operation.

Similarly, if a task issues a lock on record 1 and then issues a request to position the file at record 2, the lock manager automatically clears the lock on record 1, before positioning the file at record 2.

A lock expires when the following sequence of events occurs:

1. Its expiration time has passed since the lock was granted.
2. Another task issues a request specifying waiting that would be denied if the lock was effective. (The request is granted.)

The number of milliseconds in the lock expiration time is specified by the lock_expiration_time file attribute. Its default value is 0, meaning an unlimited expiration time. Thus, if you do not explicitly set a nonzero lock_expiration_time for the file, locks for the file cannot expire.

An expired lock is no longer effective in preventing access to the file or record by other tasks. However, it does prevent operations on the file by the task holding the expired lock.

The task holding the expired lock is prevented from any operation on the file until it clears the expired lock. This notifies the task that a lock has expired.

For example, consider the following sequence of events:

1. Task 1 is granted a 30-millisecond Preserve_Access_and_Content lock on record 1 in file 1 without automatic unlock.
2. Thirty milliseconds pass.
3. Task 1 reads record 1 from file 1. The read request restarts the expiration time count. (The lock has not yet expired because no other task has has issued a request for the record that a Preserve_Access_and_Content lock should prevent. The lock is not unlocked because automatic unlock was not requested for the lock.)
4. Thirty milliseconds pass.
5. Task 2 requests a Preserve_Content lock on record 1 in file 1. (The Task 1 lock does not expire because a Preserve_Access_and_Content lock does not prevent Preserve_Content locks.)
6. Task 3 requests, with waiting, a Preserve_Access_and_Content lock on record 1 in file 1. (The Task 1 lock expires because a Preserve_Access_and_Content lock should prevent additional Preserve_Access_and_Content locks.)

7. Task 1 attempts to read record 2 in file 1, but instead the request terminates with a nonfatal error, notifying Task 1 that it has an expired lock. Task 1 must clear the expired lock before it can successfully request any record in file 1.

Notice that in the preceding example the lock would not have expired if the lock request had specified automatic unlock.

Expired Lock Conditions

The following nonfatal conditions can be returned for an expired lock:

`aae$key_expired_lock_exists`
The operation failed due to a leftover expired lock.

`aae$auto_unlock_frustrated`
A key value could not be automatically unlocked due to an expired lock.

`aae$key_expired_lock_exists`
The key value could not be locked due to an expired lock.

`aae$expired_lock_interfered_1`
A lock with a time limit could not be changed to a lock with no time limit due to an expired lock.

`aae$expired_lock_interfered_2`
The first primary-key value in the key list for an alternate-key value could not be locked due to an expired lock. This status can be returned only if the alternate key allows duplicate values, ordered by primary key, and, while the task is waiting for the lock, another task inserts a primary-key value at the beginning of the key list.

Lock Deadlock

A deadlock is a situation in which two or more tasks need a lock already held by another task in the group of tasks. For example, the following situation is a deadlock:

- Task 1 has a lock on record 1 and needs a lock on record 2.
- Task 2 has a lock on record 2 and needs a lock on record 3.
- Task 3 has a lock on record 3 and needs a lock on record 1.

If none of the tasks releases the lock it holds, none of the tasks can complete.

A deadlock can occur either when tasks are waiting for a lock or when tasks are each repeatedly requesting a lock. The lock manager can detect the deadlock when the tasks are actually waiting for a lock; it cannot detect a deadlock when tasks are repeatedly requesting a lock.

When the lock manager receives a lock request indicating that the task wants to wait until the lock is available, it checks for a possible deadlock. To do so, it checks whether other tasks are waiting for locks held by the requesting task. If it detects a potential deadlock, it terminates the request, returning one of these nonfatal conditions.

`aae$key_deadlock`
Returned if the deadlock is with another task.

`aae$key_self_deadlock`
Returned if the deadlock is a self-deadlock (either this instance-of-open or another instance-of-open in the requesting task already has the requested lock).

To prevent a deadlock that the lock manager cannot detect, a task should limit the number of times it repeatedly requests a lock without waiting. After a fixed number of attempts, it should do one of the following:

- Issue a lock request with waiting in which case the lock manager can notify it that a potential deadlock exists.
- Assume that a potential deadlock exists and clear the locks it holds.

File Locks

Your program should request a file lock when it needs locks on many keys at the same time.

A file lock is required when your program needs more than 1024 locks at a time because 1024 is the maximum number of locks allowed for an instance of open. An attempt to exceed this limit returns the nonfatal condition aae$too_many_keylocks.

The number of locks allowed also depends on the file_limit attribute value. The lock manager tracks all locks for a file in another file called the lock file (named AAF$DEPENDENCY_FILE). The lock file size cannot exceed 90% of the file_limit value and, if an operation would cause the lock file to be more than 50% full, the operation is not allowed to begin and the fatal condition aae$lock_file_crowded is returned.

In general, the rules for using file locks are the same as those for individual locks on primary-key values. The difference is that a file lock is a lock on all primary-key values in the nested file currently selected.

A nested file cannot be deleted while any locks exist for the nested file. Locks are not discarded even when another nested file is selected.

File Lock Intents

The effect of the lock intent of a file lock is as follows:

- Exclusive_Access

  Only the owner of the lock can access records in the nested file; all requests by nonowners are denied including all lock requests.

- Preserve_Access_and_Content

  Allows Preserve_Content locks (both key locks and file locks), but denies all Exclusive_Access and Preserve_Access_and_Content locks.

- Preserve_Content

  Allows any number of Preserve_Content locks and one Preserve_Access_and_Content lock for each primary-key value and for the nested file as a whole, but denies all Exclusive_Access lock requests.

Effect of Locks on Keyed-File Calls

This section summarizes the effects of locks on calls for an open keyed file.

These calls request locks:

AMP$GET_LOCK_KEYED_RECORD	AMP$LOCK_FILE
AMP$GET_LOCK_NEXT_KEYED_RECORD	AMP$LOCK_KEY

These calls explicitly clear locks:

AMP$UNLOCK_FILE
AMP$UNLOCK_KEY

(A lock requested with automatic unlock is cleared by any call to the instance of open, except calls that fetch or store attributes or fetch access information.)

When another instance of open exists for the file, these calls require a lock on the primary-key value:

AMP$DELETE_KEY
AMP$REPLACE_KEY

A lock held by another instance of open could cause these calls to return abnormal status:

AMP$DELETE_NESTED_FILE	AMP$PUT_KEY
AMP$GET_KEY	AMP$PUT_NEXT
AMP$GET_NEXT	AMP$PUTREP
AMP$GET_NEXT_KEY	AMP$SELECT_NESTED_FILE

All other calls for an open keyed file return normal status regardless of locks.

For more information on the effect of locks on a call, see the individual call description in chapter I-3.

Creating and Deleting Alternate Keys

To create or delete alternate keys, a CYBIL program performs these steps:

1. Opens the file, if it is not already open.
2. Issues an AMP$CREATE_KEY_DEFINITION call for each alternate key to be created. Issue an AMP$DELETE_KEY_DEFINITION call for each alternate key to be deleted.
3. To implement the alternate-key definitions and deletions specified in step 2, it issues an AMP$APPLY_KEY_DEFINITIONS call. Or, to discard the specified definitions and deletions, it issues an AMP$ABANDON_KEY_DEFINITIONS call.

A program can create alternate keys in a new file or in an existing file. The point at which you should create alternate keys depends upon how the alternate key handles duplicate values.

If the file data is expected to contain duplicate values for the alternate key and the duplicate values are to be ordered first-in-first-out, the alternate key must be defined before records are written to the file. Otherwise, when the alternate index is built, the duplicate values already existing in the file are ordered by primary-key value. Duplicate values added later are ordered first-in-first-out.

If duplicate key values are not allowed for the alternate key or the duplicate values are to be ordered by primary-key value, the alternate key should be defined after records are written to the file. Building the alternate index is more efficient when the records are already in sorted order. If the alternate index is updated as each record is written, the alternate index is built in random order. This takes much longer. The efficiency difference is even greater when the file has more than one alternate index.

If the file is large, applying an alternate-key definition to a file can require considerable processing time. This is because creation of a new alternate index requires that all records in the file be read.

Using Alternate Keys

An alternate key is available for use after it has been defined and applied to the file. The following sections describe how you can use an alternate key.

In general, file access calls perform the same processing when an alternate key is selected as when the primary key is selected. The only difference is that records are accessed through the alternate index.

Record access through the alternate index means that the logical record order is the order of the alternate-key values in the alternate index. The alternate-key values are stored in ascending order.

If more than one record is associated with the same alternate-key value, the records are accessed in the order their primary-key values occur in the key list for the alternate-key value.

For example, suppose the key list for alternate-key values A and B are as follows:

A: RECORD1, RECORD3
B: RECORD2

The A records are read before the B records so that the records would be read sequentially: RECORD1, RECORD3, RECORD2.

Selecting an Alternate Key

When a keyed file is opened, the system assumes that file processing is by primary key. That is, the selected key is initially the primary key. You can change the selected key by calling AMP$SELECT_KEY. The call specifies the name of the key to be selected.

An AMP$SELECT_KEY call specifies the name of the key as it was defined when the key was created. To specify the primary key on an AMP$SELECT_KEY call, specify the name $PRIMARY_KEY.

The key selected by an AMP$SELECT_KEY call is used until another AMP$SELECT_KEY call changes the selected key or until the file is closed.

File Positioning After Alternate-Key Selection

When an AMP$SELECT_KEY call selects a different key, it sets the file position to the beginning of the index for that key. (If the key specified on an AMP$SELECT_KEY call is already the selected key, the file position is not changed.) After an alternate key is selected, all file positioning follows the logical record order represented in the alternate index.

As described earlier in this chapter, several calls are available to position a keyed file. Those calls that both position the file and read and write data are described later. The following calls position the file without reading or writing data:

AMP$START
: Positions the file to access the record having the specified value for the selected key.

AMP$REWIND
: Positions the file at the beginning of the index for the selected key. The file is positioned to access the record with the lowest value for the selected key.

AMP$SKIP
: Positions the file forward or backward the specified number of records (according to the record order provided by the index for the selected key).

Reading Records After Alternate-Key Selection

In general, the calls to read (or get) a record perform the same when an alternate key is selected as when the primary key is selected. The only difference is that records are accessed through the alternate index.

Random get calls specify the record to be read by its alternate-key value. Sequential get calls access records in sorted order by alternate-key value.

These calls get a record and position the file to read or write the next record. The next record is the record having the next primary-key value listed in the alternate index.

AMP$GET_KEY
: Gets the first record in the key list of the specified alternate-key value and positions the file to read the next record.

  An AMP$GET_KEY call specifies the alternate-key value either in the location referenced by the key_location pointer or (with a NIL key_location pointer) in the working storage area. The second method is especially useful for concatenated alternate keys because the fields of the key can be assembled in the working storage area. Each key field value is stored in the working storage area at its actual position within the record.

AMP$GET_NEXT_KEY
Gets the record at the current position in the alternate index, returns the alternate-key value of the record read, and positions the file to read the next record.

The alternate-key value returned is the value stored in the alternate index. If the alternate-key type is AMC$COLLATED_KEY, the key values are stored in collated form. In collated form, each character is represented by the lowest character code having the same collating weight.

For example, assume that lowercase letters are collated as equal to the corresponding uppercase letters (each uppercase/lowercase pair has the same collating weight). Then the alternate-key value is stored (and later returned) using only uppercase letters.

AMP$GET_NEXT
Gets the record at the current position in the alternate index and positions the file to read the next record.

Updating an Alternate Index

A call to put, replace, or delete a record cannot specify an alternate-key value; a key value specified on a put, replace, or delete call is expected to be a primary-key value even if an alternate key is currently selected. However, put, replace, and delete calls do update any alternate indexes affected by the operation.

When a call deletes a record in the file, any alternate index entries for the record are deleted.

When a call writes a new record to the file, an entry for the record is added to the alternate indexes (unless the record is excluded from an index by sparse-key control). The new record can then be read by its alternate-key value.

When a call replaces an existing record in the file, the alternate index entries for the record are replaced with the appropriate entries for the new record. (The alternate-key value could have changed or sparse-key control could exclude the record from an alternate index.)

To update an alternate index, the file must be open for modify, shorten, and append access.

If an alternate index in the file was created using the default duplicate_key_ control value AMC$NO_DUPLICATES_ALLOWED, a record having the same alternate-key value as a record already in the file cannot be written to the file. An attempt to put or replace a duplicate record does not write the record and returns a nonfatal error.

Fetching Access Information After Alternate-Key Selection

An AMP$FETCH_ACCESS_INFORMATION call can return the following items of information. (The call format is in the CYBIL File Management manual.) This list highlights the meaning of each item when returned immediately after a call that specifies an alternate-key value:

duplicate_value_inserted
: Boolean indicating whether the last AMPPUT, AMPPUTREP, AMP$REPLACE, or AMP$APPLY_KEY_DEFINIIONS call detected a duplicate alternate-key value.

  The duplicate_value_inserted item does not identify the duplication. An AMPPUT, AMPPUTREP, or AMP$REPLACE call can detect a duplicate value for any alternate key in the file that allows duplicates. An AMP$APPLY_KEY_DEFINITIONS call can detect a duplicate value for any record in the file.

file_position
: Returns the current file position as described later under File Position Returned.

primary_key
: Primary-key value of the record at the current file position (the next record).

NOTE

The AMP$FETCH_ACCESS_INFORMATION call must specify a pointer to the location where the primary-key value is to be returned. The pointer must be specified in the PRIMARY_KEY field in the array specified by the fetch_items parameter.

selected_key_name
: Name of the currently selected key. If the primary key is currently selected, the name $PRIMARY_KEY is returned.

File Position Returned

At completion of each AMP$START, AMP$GET_KEY, or AMP$GET_NEXT_KEY call, a value is returned in the file_position variable. The value returned is AMCEOR, AMCEOI, or AMC$END_OF_KEY_LIST as shown in the following table.

Table I-2-1. File Position Returned

	AMC$EOR	AMC$END_OF_KEY_LIST	AMC$EOI
AMP$START	Not applicable.	The alternate index is positioned at the end of a key list and at the beginning of the next key list. The next keylist is for either the specified alternate-key value or the next higher alternate-key value if the specified value was not found.	The alternate index is positioned at its end because the specified alternate-key value was higher than any alternate-key value in the index.
AMP$GET_KEY	A record associated with the alternate-key value has been returned, and if an AMP$GET_NEXT_KEY call were issued next, it would return the next record in the key list for the same alternate-key value.	The last (or only) record associated with the alternate-key value has been returned, and if an AMP$GET_NEXT_KEY call were issued next, it would return a record with another alternate key value or the file_position AMC$EOI.	Same as for AMP$START.
AMP$GET_NEXT_KEY	Same as for AMP$GET_KEY.	Same as for AMP$GET_KEY.	No record is returned because the file is positioned at the end of the alternate index.

Retrieving Alternate-Index Information

An alternate index is a structure independent from the file data. Thus, a program can fetch information from the alternate index without requiring access to the file data. This section describes the calls that fetch information from the alternate index.

An AMP$GET_KEY_DEFINITIONS call retrieves the definitions of existing alternate keys. Your program could use the definitions returned by AMP$GET_KEY_DEFINITIONS to:

- Determine the attributes of an alternate key
- Define identical or similar alternate keys in another file

For example, you may want to get the alternate-key definitions from an old file to apply to a re-created file.

An AMP$GET_NEXT_PRIMARY_KEY_LIST retrieves primary-key values from the alternate index. The primary-key values are returned in the order the values are stored in the alternate index, beginning at the current position.

Generally, AMP$GET_PRIMARY_KEY_COUNT and AMP$GET_SPACE_USED_FOR_KEY calls prepare for subsequent calls that read or position by alternate key. AMP$GET_PRIMARY_KEY_COUNT counts the number of primary-key values for a range of alternate-key values in the alternate index. AMP$GET_SPACE_USED_FOR_KEY counts the number of alternate-index blocks that contain the specified alternate-key value range.

AMP$GET_PRIMARY_KEY_COUNT gives the program the exact number of primary-key values it would receive if it calls AMP$GET_NEXT_PRIMARY_KEY_LIST for the alternate-key value range. To count the values, AMP$GET_PRIMARY_KEY_COUNT sequentially reads the alternate-index records that contain the information.

AMP$GET_SPACE_USED_FOR_KEY does not actually read the alternate-index records that contain the primary-key values. It just counts the blocks that would contain the records for a given range of alternate-key values. This is much faster. The count returned is generally used to compare with a count returned by another AMP$GET_SPACE_USED_FOR_KEY to determine the shorter primary-key value list.

As an example of a use for this call, assume that a program is to find a set of records in response to this query.

Find the Jones on Madison Avenue with more than two dependents.

Assume that both Jones and Madison Avenue are alternate-key values, but number of dependents is not. The program must actually read the data records to determine the number of dependents.

The program could read the set of records for either Jones or Madison Avenue. To minimize the number of data records it must read, it should fetch the shorter list. To determine which is the shorter list, it could compare values returned by either AMP$GET_PRIMARY_KEY_COUNT or AMP$GET_SPACE_USED_FOR_KEY calls. When the exact number of primary-key values is not needed, it is faster to call AMP$GET_SPACE_USED_FOR_KEY.

AMP$GET_SPACE_USED_FOR_KEY returns two values, block_count and block_space. The block_space value is the block_count value multiplied by the block size for the file. When comparing sets of records from more than one file, a program should compare the block_space value returned, instead of the block_count values. The block_space value is more useful in this case because the block size could differ in the files.

Program Examples

This section contains CYBIL program examples that perform these functions:

- Create an indexed-sequential file
- Update an indexed-sequential file
- Create and use an alternate key
- Create and delete nested files

Indexed-Sequential File Creation Example

This program (module CREATE) creates an indexed-sequential file named INDEXED by copying records from a sequential file with local file name ORIGINAL_DATA. The first 15 characters of the record are used as the embedded primary key. The records are fixed-length records, each 55 characters long.

The following is a listing of the data in file ORIGINAL_DATA. The first column is a country name, the second is the population of the country, the third is the size of the country (in square miles), and the fourth is the capital of the country. There are several errors in the data; these will be fixed by the second example program.

```
Algeria            19709000    919591 Algiers
Australia          14796000   2967895 Melbourne
Austria             7476000     32374 Vienna
Belgium             9875000     11781 Brussels
Canada             20050000   3851791 Montreal
Denmark             5157000     16629 Copenhagen
France             53844000    211207 Paris
Great Britain      55717000     94226 London
India             700734000   1269340 Delhi
Ireland             3349000     27136 Dublin
Ivory Coast         8513000    124503 Abidjan
Japan             118783000    143750 Yokohama
Mexico             70143000    761601 Mexico
Sweden              8335000    173731 Stockholm
Switzerland         6300000     15941 Bern
Tanzania           18744000    364898 Zanzibar
Turkey             47284000    301381 Ankara
United Kingdom     55717000     94226 London
United States     225195000   3615105 Washington
USSR              269302000   8649498 Moscow
Venezuela          15771000    352143 Caracas
West Germany       60948000     95976 Bonn
```

This is a source listing of the program that creates the indexed-sequential file. The program uses the common procedures listed in appendix E to inspect the status variable after each call and to produce a report on file $OUTPUT.

```
MODULE create ;

{ This program creates an indexed sequential file (ISFILE) from }
{ a sequential file (DATAIN).  The primary key for ISFILE is    }
{ the name of the country.                                      }
```

```
  CONST
    key_length = 15,
    max_record_length = 55,
    record_count = 30,
    key_position = 0,
    data_padding = 15,
    index_padding = 10,
    index_levels = 2;

  VAR
{ Declare variables for ISFILE.}
    isfile: amt$local_file_name,
    isfile_id: amt$file_identifier,
    isfile_fpos: amt$file_position,
{ Declare variables for DATAIN.}
    datain: amt$local_file_name,
    sqfile_id: amt$file_identifier,
    sqfile_fpos: amt$file_position,
    sqfile_transfer_count: amt$transfer_count,
    sqfile_byte_address: amt$file_byte_address,
{ Wsa is used by both ISFILE and DATAIN.}
    wsa: string (max_record_length);

{ Establish for file_description an array of file attribute }
{ values.                                                   }

  VAR file_description:  [STATIC] array [1 ..  13] of
    amt$file_item :=
     [[amc$file_organization,      amc$indexed_sequential],
      [amc$max_record_length,      max_record_length],
      [amc$record_type,            amc$ansi_fixed],
      [amc$average_record_length,  max_record_length],
      [amc$embedded_key,           TRUE],
      [amc$key_length,             key_length],
      [amc$key_position,           key_position],
      [amc$key_type,               amc$uncollated_key],
      [amc$data_padding,           data_padding],
      [amc$index_padding,          index_padding],
      [amc$index_levels,           index_levels],
      [amc$estimated_record_count, record_count],
      [amc$message_control,        $amt$message_control
                                    [amc$trivial_errors,
                                     amc$messages,
                                     amc$statistics]]];
```

```
  PROGRAM creation_phase (VAR program_status : ost$status) ;

    p#start_report_generation(
      'Begin indexed-sequential file creation.');
    isfile := 'indexed';
    datain := 'original_data';
    amp$file (isfile, file_description, status);
       p#inspect_status_variable ;

    amp$open (isfile, amc$record, NIL, isfile_id, status);
      p#inspect_status_variable ;
    amp$open (datain, amc$record, NIL, sqfile_id, status);
      p#inspect_status_variable ;

{ The next part of the program reads records from DATAIN and  }
{ writes the records to ISFILE.  A WHILE loop is used to read }
{ and write the records until the file position of DATAIN is  }
{ end-of-information.                                         }

    wsa := ' ';
    amp$get_next (sqfile_id, ^wsa, max_record_length,
      sqfile_transfer_count, sqfile_byte_address,
      sqfile_fpos, status);
      p#inspect_status_variable;

    WHILE (sqfile_fpos <> amc$eoi) DO
      {The working storage length (the third parameter) is
      {ignored because the record type is amc$ansi_fixed.
      amp$put_key (isfile_id, ^wsa, 0, NIL, osc$wait, status);
        p#inspect_status_variable;
      wsa := ' ';
      amp$get_next (sqfile_id, ^wsa, max_record_length,
        sqfile_transfer_count, sqfile_byte_address,
        sqfile_fpos, status);
        p#inspect_status_variable;
    WHILEND;

    amp$close (isfile_id, status);
      p#inspect_status_variable;
    amp$close (sqfile_id, status);
      p#inspect_status_variable;

    p#stop_report_generation(
      'Indexed-sequential file creation complete.');
    program_status.normal := TRUE ;
    { Exit with normal status. }

  PROCEND creation_phase ;
```

```
?? PUSH (LIST := OFF) ??
{ This deck contains the common procedures listed in appendix E. }
*copyc comproc

*copyc amp$close
*copyc amp$file
*copyc amp$get_next
*copyc amp$open
*copyc amp$put_key
?? POP ??

MODEND create;
```

Assuming the program source text is stored as file $USER.CREATE, the following are the SCL commands required to expand, compile, attach the data files, and execute the program. After the commands is a listing of the statistical messages from the program.

```
/create_source_library base=temporary_library
/scu base=temporary_library
sc/create_deck deck=create modification=original ..
sc../source=$user.create
sc/expand_deck deck=create ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit, write_library=no
/cybil input=compile list=listing
/attach_file $user.original_data
/lgo

Begin indexed-sequential file creation.
--  File INDEXED :  0 DELETE_KEYs done since last open.
--  File INDEXED :  0 GET_KEYs done since last open.
--  File INDEXED :  0 GET_NEXT_KEYs done since last open.
--  File INDEXED :  22 PUT_KEYs (and PUTREPs->put) since last
open.
--  File INDEXED :  0 PUTREPs done since last open.
--  File INDEXED :  0 REPLACE_KEYs (and PUTREPs->replace) since
last open.
No error has been found by the program.
Indexed-sequential file creation complete.
```

Indexed-Sequential File Update Example

This program (module UPDATE) adds, deletes, and replaces records in the file INDEXED created by the CREATE program. The program reads its input from a file named UPDATE_DATA.

The directives on file UPDATE_DATA are listed in the program output. In the program, only the first letter of the words Delete, Replace, and Put are used. The full word is included in the file to make the example clearer. Only the primary key is required to delete a record.

This is a source listing of the program that updates the indexed-sequential file. The program uses the common procedures listed in appendix E to inspect the status variable after each call and to produce a report on file $OUTPUT.

```
MODULE update;

{ This program updates an indexed sequential file (INDEXED) }
{ using information in an update file (UPDATE_DATA).        }

CONST
  record_length = 55;

VAR
{ Declare variables for ISFILE.}
  isfile: amt$local_file_name := 'indexed',
  isfile_id: amt$file_identifier,
  isfile_fpos: amt$file_position,
  key: string (15),
  isfile_wsa: string (record_length),

{ Declare variables for UPDATE.}
  update: amt$local_file_name := 'update_data',
  update_id: amt$file_identifier,
  update_fpos: amt$file_position,
  update_transfer_count: amt$transfer_count,
  update_byte_address: amt$file_byte_address,
  update_wsa: string (record_length + 7),

{ Declare access_selections array for amp$open.}
  access_selections:  [STATIC] array [1 ..  1] of amt$file_item
    := [[amc$message_control, $amt$message_control
      [amc$trivial_errors, amc$messages, amc$statistics]]];
```

```
  PROGRAM updating_phase (VAR program_status : ost$status) ;

    p#start_report_generation('Begin file update.');
    amp$open (isfile, amc$record, ^access_selections,
      isfile_id, status);
      p#inspect_status_variable;
    amp$open (update, amc$record, NIL, update_id, status);
      p#inspect_status_variable;

{ The WHILE loop that follows reads an update record from UPDATE  }
{ and edits ISFILE accordingly.  The update information is        }
{ contained in the first 7 characters of the records in UPDATE;   }
{ however, only the first character is used to determine          }
{ whether a delete, put, or replace operation is to be            }
{ performed.  If the operation requested is not a delete, put, or }
{ replace, a message and the update record are printed on the     }
{ output listing.  If the status parameter check shows that an    }
{ error occurred, then control is returned to the system.         }

   update_wsa := ' ';
   amp$get_next (update_id, ^update_wsa, STRLENGTH(update_wsa),
      update_transfer_count, update_byte_address, update_fpos,
      status);
      p#inspect_status_variable;
   WHILE (update_fpos <> amc$eoi) DO
     p#put_m (TRUE, update_wsa(1, update_transfer_count));
     isfile_wsa := update_wsa (8, * );
     key := isfile_wsa (1, 15);
     CASE update_wsa (1) OF
     = 'D' =
       amp$delete_key (isfile_id, ^key, osc$wait, status);
       p#inspect_status_variable ;
     = 'P', 'R' =
       amp$putrep (isfile_id, ^isfile_wsa, 0, NIL, osc$wait,
                   status);
       p#inspect_status_variable;
     ELSE
       p#put_m (FALSE, 'Invalid code given as first character. ');
       p#put_m (TRUE , update_wsa(1, update_transfer_count));
    CASEND;
  update_wsa (1, * ) := ' ';
  amp$get_next (update_id, ^update_wsa, STRLENGTH(update_wsa),
    update_transfer_count, update_byte_address,
    update_fpos, status);
    p#inspect_status_variable;
   WHILEND;
```

```
    amp$close (isfile_id, status);
      p#inspect_status_variable;
    amp$close (update_id, status);
      p#inspect_status_variable;

     p#stop_report_generation('File update complete.');
     program_status.normal := TRUE ;
     { Exit with normal status. }

  PROCEND updating_phase ;

?? PUSH (LIST:=OFF) ??
{ The COMPROC deck contains the common procedures listed in }
{ appendix E. }
*copyc comproc

*copyc amp$close
*copyc amp$delete_key
*copyc amp$get_next
*copyc amp$open
*copyc amp$putrep
?? POP ??

MODEND update;
```

Assuming the program source text is stored on file $USER.UPDATE, the following are the SCL commands required to expand, compile, attach the data file, and execute the program. It is assumed that the indexed-sequential file to be updated is accessible as file INDEXED in the $LOCAL catalog. After the commands is a listing of the statistical messages from the file update program.

```
/create_source_library base=temporary_library
/scu base=temporary_library
sc/create_deck deck=update modification=original ..
sc../source=$user.update
sc/expand_deck deck=update ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit, write_library=no
/cybil input=compile list=listing
/attach_file $user.update_data
/lgo

Begin file update.
ReplaceCanada                24336000   3851791 Ottawa
Put    China               1053788000   3705390 Beijing
Delete Great Britain
Put    Spain                 38686000    194897 Madrid
Put    Italy                 57513000    116303 Rome
ReplaceJapan                 11878300    143750 Tokyo
--  File INDEXED :  1 DELETE_KEYs done since last open.
--  File INDEXED :  0 GET_KEYs done since last open.
--  File INDEXED :  0 GET_NEXT_KEYs done since last open.
--  File INDEXED :  3 PUT_KEYs (and PUTREPs->put) since last
open.
--  File INDEXED :  5 PUTREPs done since last open.
--  File INDEXED :  2 REPLACE_KEYs (and PUTREPs->replace) since
last open.
No error has been found by the program.
File update complete.
```

Alternate-Key Example

The following program illustrates the use of alternate keys. The program uses the indexed-sequential file created and updated in the earlier examples in this chapter. It also uses the common procedures listed in appendix E.

The program defines the capital field as the alternate key field. It then copies the records to file ALTERNATE_KEY_OUTPUT, sorted by the alternate key.

This is a source listing of the program.

```
MODULE example_3 ;

{This module defines and then uses alternate keys for ISFILE.}

CONST
  max_record_length = 55;

VAR
{ Declare variables for ISFILE.}
  isfile: amt$local_file_name,
  isfile_id: amt$file_identifier,
  isfile_fpos: amt$file_position,

{Declare variables for alternate key CAPITAL_KEY.}
  capital_key_name: amt$key_name :=  'capital_key',
  capital_key_position: amt$key_position := 41,
  capital_key_length: amt$key_length := 14,

{ Declare variables for SQFILE.}
  sqfile: amt$local_file_name,
  sqfile_id: amt$file_identifier,
  sqfile_byte_address: amt$file_byte_address;

VAR
  wsa: string(max_record_length),
  record_length : amt$max_record_length;

{ Declare access_selections array for amp$open of ISFILE.}
VAR
  access_selections_isfile: [STATIC] array [1 .. 1] of
     amt$file_item :=
    [[amc$message_control, $amt$message_control
      [amc$trivial_errors, amc$messages, amc$statistics]]];
```

```
{ Establish the file attribute array for file_description.}
VAR
  file_description: [STATIC] array [1 .. 2] of amt$file_item :=
    [[amc$file_organization,     amc$sequential],
    [amc$max_record_length,      max_record_length]];


{ Declare access_selections array for amp$open of SEQFILE.}
VAR
  access_selections_sqfile: [STATIC] array [1 .. 1]
    of amt$file_item :=
    [[amc$file_contents,       amc$legible]];

VAR
  capital_attributes: [STATIC,READ] array [1..1]
    of amt$optional_key_attribute :=
    [[amc$duplicate_keys, amc$ordered_by_primary_key]];


  PROGRAM alternate_key_phase (VAR program_status : ost$status);

    p#start_report_generation('Begin alternate keys example.');

{These calls specify file attributes and open files. }
   isfile := 'indexed';
   sqfile := 'alternate_key_output';
   amp$file (sqfile, file_description, status);
     p#inspect_status_variable;
   amp$open (isfile, amc$record, ^access_selections_isfile,
     isfile_id, status);
     p#inspect_status_variable;
   amp$open (sqfile, amc$record, ^access_selections_sqfile,
     sqfile_id, status);
     p#inspect_status_variable;

{These calls define and generate the alternate index. }
   amp$create_key_definition (isfile_id, capital_key_name,
     capital_key_position, capital_key_length,
     ^capital_attributes, status);
     p#inspect_status_variable;
   amp$apply_key_definitions (isfile_id, status);
      p#inspect_status_variable;
```

```
{These calls select the alternate key and read the first record. }
   amp$select_key (isfile_id, capital_key_name, status);
     p#inspect_status_variable;
   amp$get_next_key (isfile_id, ^wsa, max_record_length, NIL,
     record_length, isfile_fpos, osc$wait, status);
     p#inspect_status_variable ;

{ This loop copies the records in the indexed-sequential }
{ file to the sequential file in the order the records   }
{ are referenced in the alternate index.                 }
   WHILE (isfile_fpos <> amc$eoi) DO
     amp$put_next (sqfile_id, ^wsa, max_record_length,
       sqfile_byte_address, status);
       p#inspect_status_variable;
     wsa (1, * ) := ' ';
     amp$get_next_key (isfile_id, ^wsa, max_record_length, NIL,
       record_length, isfile_fpos, osc$wait, status);
       p#inspect_status_variable ;
   WHILEND;

   amp$close (isfile_id, status);
     p#inspect_status_variable;
   amp$close (sqfile_id, status);
     p#inspect_status_variable ;

    p#stop_report_generation('Alternate keys example complete.');
    program_status.normal := TRUE;
    { Exit with normal status. }

  PROCEND alternate_key_phase;

?? PUSH (LIST:=OFF) ??
{ This deck contains the common procedures listed in appendix E. }
*copyc comproc

*copyc amp$apply_key_definitions
*copyc amp$close
*copyc amp$create_key_definition
*copyc amp$file
*copyc amp$get_next_key
*copyc amp$open
*copyc amp$put_next
*copyc amp$select_key
?? POP ??
MODEND example_3 ;
```

Assuming the source program is stored as deck ALTERNATE_KEYS on source library file $USER.MY_LIBRARY, the following is a listing of the SCL commands required to expand, compile and execute the program. It is assumed that the indexed-sequential file is accessible as file INDEXED in the $LOCAL catalog.

```
/scu base=$user.my_library
sc../expand_deck deck=(alternate_keys) ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit, write_library=no
/cybil input=compile
/lgo

Begin alternate keys example.
--  File INDEXED : begin creating labels for alternate key
                   definitions.
--  File INDEXED : finished creating labels for alternate key
                   definitions.
--  File INDEXED : begin the data pass that collects alternate
                   key values.
--  File INDEXED : AMP$APPLY_KEY_DEFINITIONS has reached a file
                   boundary : EOI.
--  File INDEXED : data pass completed.
--  File INDEXED : begin sorting the alternate key values.
--  File INDEXED : sorting completed.
--  File INDEXED : begin building alternate key indexes into
                   the file.
--  File INDEXED : completed building the indexes into the file.
--  File INDEXED : AMP$GET_NEXT_KEY has reached a file
                   boundary : EOI.
--  File INDEXED : 0 DELETE_KEYs done since last open.
--  File INDEXED : 0 GET_KEYs done since last open.
--  File INDEXED : 48 GET_NEXT_KEYs done since
                   last open.
--  File INDEXED : 0 PUT_KEYs (and PUTREPs->put) since last open.
--  File INDEXED : 0 PUTREPs done since last open.
--  File INDEXED : 0 REPLACE_KEYs (and PUTREPs->replace) since
                   last open.
No error has been found by the program.
Alternate keys example complete.
```

This is a listing of the ALTERNATE_KEY_OUTPUT file written by the program.

```
Ivory Coast          8513000   124503 Abidjan
Algeria             19709000   919591 Algiers
Turkey              47284000   301381 Ankara
China             1053788000  3705390 Beijing
Switzerland         6300000    15941 Bern
West Germany       60948000    95976 Bonn
Belgium             9875000    11781 Brussels
Venezuela          15771000   352143 Caracas
Denmark             5157000    16629 Copenhagen
India             700734000  1269340 Delhi
Ireland             3349000    27136 Dublin
United Kingdom     55717000    94226 London
Spain              38686000   194897 Madrid
Australia          14796000  2967895 Melbourne
Mexico             70143000   761601 Mexico
USSR              269302000  8649498 Moscow
Canada             24336000  3851791 Ottawa
France             53844000   211207 Paris
Italy              57513000   116303 Rome
Sweden              8335000   173731 Stockholm
Japan              11878300   143750 Tokyo
Austria             7476000    32374 Vienna
United States     225195000  3615105 Washington
Tanzania           18744000   364898 Zanzibar
```

Nested File Example

This example is a CYBIL program that first copies the nested-file definitions from one keyed file to another keyed file and then destroys the original nested files.

The program copies the nested-file definitions from file EXISTING_KEYED_FILE to file ANOTHER_KEYED_FILE.

```
MODULE nested_file_module;

  VAR
    lfn1: [STATIC] amt$local_file_name :=
                   'existing_keyed_file',
    lfn2: [STATIC] amt$local_file_name :=
                   'another_keyed_file',
    fid1: amt$file_identifier,
    fid2: amt$file_identifier,
    access_information_ptr: ^amt$access_information,
    definitions_ptr: ^amt$nested_file_definitions,
    nested_file_count: amt$nested_file_count,
    element: amt$nested_file_count;

{ This program copies the nested-file definitions in file
{ EXISTING_KEYED_FILE (LFN1) to file ANOTHER_KEYED_FILE (LFN2).
{ It then deletes all nested files (except $MAIN_FILE) from
{ LFN1.  Any data in the LFN1 nested files (other than in
{ $MAIN_FILE) is discarded.

PROGRAM nested_file_example (VAR program_status: ost$status);

  p#start_report_generation(
    'Start copying of nested-file definitions.');

  amp$open(lfn1, amc$record, NIL, fid1, status);
    p#inspect_status_variable;

{ These statements fetch the number of nested files currently
{ defined in LFN1.

  ALLOCATE access_information_ptr : [1..1];
  access_information_ptr^[1].key:=amc$number_of_nested_files;
  amp$fetch_access_information(fid1, access_information_ptr^,
    status);
    p#inspect_status_variable;
```

```
{ A value is returned for number_of_nested_files only if the
{ file is a keyed file.  If it is not, the program sends a
{ message and terminates.

    IF NOT access_information_ptr^[1].item_returned THEN
      p#stop_report_generation(
        'File EXISTING_KEYED_FILE is not a keyed file.');
      pmp$exit( status );
    IFEND;

{ This statement allocates an array large enough to hold a
{ nested-file definition record for each nested file in LFN1.

  ALLOCATE definitions_ptr :
      [1..access_information_ptr^[1].number_of_nested_files];
  amp$get_nested_file_definitions(fid1, definitions_ptr^,
     nested_file_count, status);
    p#inspect_status_variable;

  amp$open(lfn2, amc$record, NIL, fid2, status);
    p#inspect_status_variable;

{ This loop defines each nested file from LFN1 in LFN2.
{ Element 1 of the array is skipped because it contains the
{ definition of nested file $MAIN_FILE which already exists.

  /define_loop/
  FOR element := 2 TO nested_file_count DO

    amp$create_nested_file(fid2,
      definitions_ptr^[element], status);

    IF NOT status.normal THEN
      IF status.condition=ame$unimplemented_request THEN
        p#put_m (TRUE,
          'File ANOTHER_KEYED_FILE is not a keyed file.');
        EXIT /define_loop/;
      ELSE
        p#inspect_status_variable;
      IFEND;
    IFEND;

  FOREND /define_loop/;

  amp$close(fid2, status);
    p#inspect_status_variable;
```

```
  p#put_m (TRUE, 'Nested file definition copying is done.');
  p#put_m (TRUE, 'Nested-file deletion now begins.');

{ This loop deletes each nested file in LFN1.  Element 1 in
{ the array is skipped because it contains the definition
{ of nested file $MAIN_FILE which cannot be deleted.

  FOR element := 2 TO nested_file_count DO

    amp$delete_nested_file(fid1,
      definitions_ptr^[element].nested_file_name, status);
    p#inspect_status_variable;

  FOREND;

  amp$close(fid1, status);
    p#inspect_status_variable;

  p#stop_report_generation(
    'Nested-file deletion complete.');

PROCEND nested_file_example;

?? PUSH (LIST := OFF) ??
{ The COMPROC deck contains the common
{ procedures listed in appendix E.
*copyc comproc

*copyc amp$open
*copyc amp$fetch_access_information
*copyc amp$get_nested_file_definitions
*copyc amp$delete_nested_file
*copyc amp$close
*copyc amp$create_nested_file

{ This directive is required to copy the
{ named condition identifier declaration.
*copyc ame$unimplemented_request
?? POP ??

MODEND nested_file_module
```

The following commands prepare the program for execution.

```
"Commands to expand the program text.
create_source_library result=temporary_library
scu, base=temporary_library
  create_deck, deck=nested_file_program, ..
    modification=original, source=$user.nested_file_program
  create_deck, deck=comproc, modification=original, ..
    source=$user.appendix_E_procedures
  expand_deck, deck=nested_file_program, ..
    alternate_base=($system.cybil.osf$program_interface, ..
      $system.common.psf$external_interface_source)
  quit, write_library=no
"
"Command to compile the expanded text on file COMPILE.
cybil, input=compile, list=listing
```

After execution of the preceding commands, the object program is on file LGO, ready for execution. To demonstrate program execution, the files $USER.INDEXED_SEQUENTIAL_FILE and $USER.DIRECT_ACCESS_FILE are copied to local files with the correct names. (Or, the files could be attached as local files with these names, in which case, the program would change the original files instead of copies.)

```
/copy_keyed_file, input=$user.indexed_sequential_file, ..
../output=existing_keyed_file
/copy_keyed_file, input=$user.direct_access_file, ..
../output=another_keyed_file
/lgo

Start copying of nested-file definitions.
No error has been found by the program.
Nested file definition copying is done.
Nested-file deletion now begins.
No error has been found by the program.
Nested-file deletion complete.
/
```

You could confirm that the nested files have been copied and deleted by entering these commands:

```
display_keyed_file_properties, existing_keyed_file
display_keyed_file_properties, another_keyed_file
```

(The COPY_KEYED_FILE and DISPLAY_KEYED_FILE_PROPERTIES commands are described in the SCL Advanced File Management Usage manual.)

Keyed-File Interface Calls I-3

Using Keyed-File Interface Calls I-3-1
File Access I-3-2
AMP$ABANDON_KEY_DEFINITIONS I-3-4
AMP$APPLY_KEY_DEFINITIONS I-3-5
AMP$CREATE_KEY_DEFINITION I-3-7
AMP$CREATE_NESTED_FILE I-3-14
AMP$DELETE_KEY I-3-17
AMP$DELETE_KEY_DEFINITION I-3-19
AMP$DELETE_NESTED_FILE I-3-20
AMP$GET_KEY I-3-22
AMP$GET_KEY_DEFINITIONS I-3-27
AMP$GET_LOCK_KEYED_RECORD I-3-30
AMP$GET_LOCK_NEXT_KEYED_RECORD I-3-34
AMP$GET_NESTED_FILE_DEFINITIONS I-3-38
AMP$GET_NEXT_KEY I-3-40
AMP$GET_NEXT_PRIMARY_KEY_LIST I-3-43
AMP$GET_PRIMARY_KEY_COUNT I-3-47
AMP$GET_SPACE_USED_FOR_KEY I-3-51
AMP$LOCK_FILE I-3-54
AMP$LOCK_KEY I-3-56
AMP$PUT_KEY I-3-59
AMP$PUTREP I-3-62
AMP$REPLACE_KEY I-3-64
AMP$SELECT_KEY I-3-66
AMP$SELECT_NESTED_FILE I-3-67
AMP$START I-3-69
AMP$UNLOCK_FILE I-3-72
AMP$UNLOCK_KEY I-3-73

Keyed-File Interface Calls I-3

This chapter contains detailed descriptions of each keyed-file interface call, organized in alphabetical order by the procedure name.

NOTE

As described in the manual introduction, a CYBIL program must include a *COPYC directive for each keyed-file interface procedure call it uses.

When you expand your program, you must specify these files as alternate base libraries:

$SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE
$SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE

When you execute your program, you must add the following object library file to the program library list:

$LOCAL.AAF$44D_LIBRARY

Using Keyed-File Interface Calls

When using keyed-file interface calls, you follow the same general rules you follow when using the other file interface calls described in the CYBIL File Management and CYBIL Sequential and Byte Addressable Files manuals.

A keyed-file interface call can only be issued for an instance of open of a keyed file. As shown in the individual descriptions, each call references a file by the file identifier returned by the AMP$OPEN call that opened the file. The AMP$OPEN call is described in detail in the CYBIL File Management manual.

File processing is guided by the attribute values of the file. The file attributes used by keyed files are described in chapter I-4. The calls that specify file attribute values are described in detail in the CYBIL File Management manual.

File Access

You can use a file only if you have access to it. Your access to a file is limited by the permissions you have been granted to the file. You can limit access further by requesting a subset of your permitted access modes when attaching the file. This process is described in the SCL System Interface Usage manual.

The access allowed for a particular instance of open is limited by the access_mode file attribute as specified when the file is opened. The following is a list of the access modes required for each keyed-file interface call.

Call	Access Modes Required
AMP$ABANDON_KEY_DEFINITIONS	Append, shorten, and modify
AMP$APPLY_KEY_DEFINITIONS	Append, shorten, and modify
AMP$CREATE_KEY_DEFINITION	Append, shorten, and modify
AMP$CREATE_NESTED_FILE	Append, shorten, and modify
AMP$DELETE_KEY	Shorten
AMP$DELETE_KEY_DEFINITION	Append, shorten, and modify
AMP$DELETE_NESTED_FILE	Append, shorten, and modify
AMP$GET_KEY	Read (modify required to record statistics)
AMP$GET_KEY_DEFINITIONS	Any access mode
AMP$GET_LOCK_KEYED_RECORD	Read (modify required to record statistics)
AMP$GET_LOCK_NEXT_KEYED_RECORD	Read (modify required to record statistics)
AMP$GET_NESTED_FILE_DEFINITIONS	Any access mode
AMP$GET_NEXT_KEY	Read (modify required to record statistics)
AMP$GET_NEXT_PRIMARY_KEY_LIST	Read
AMP$GET_PRIMARY_KEY_COUNT	Read
AMP$GET_SPACE_USED_FOR_KEY	Read
AMP$LOCK_FILE	Any access mode
AMP$LOCK_KEY	Any access mode
AMP$PUT_KEY	Append (shorten and modify also required if the file has one or more alternate keys)

Call	Access Modes Required
AMP$PUTREP	Append and shorten (modify also required if the file has one or more alternate keys)
AMP$REPLACE_KEY	Append and shorten (modify also required if the file has one or more alternate keys)
AMP$SELECT_NESTED_FILE	Any access mode
AMP$SELECT_KEY	Any access mode
AMP$START	Read
AMP$UNLOCK_FILE	Any access mode
AMP$UNLOCK_KEY	Any access mode

AMP$ABANDON_KEY_DEFINITIONS

Purpose Discards the pending alternate-key definitions or deletions.

Format

```
AMP$ABANDON_KEY_DEFINITIONS
(file_identifier,status);
```

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$no_definitions_pending
aae$not_enough_permission

Remarks

- A pending alternate-key definition or deletion is one that has been requested but has not yet been discarded or applied to the nested file. An AMP$ABANDON_KEY_DEFINITIONS call or the closing of the file discards all pending definitions and deletions. An AMP$APPLY_KEY_DEFINITIONS call applies all pending definitions and deletions.
- AMP$ABANDON_KEY_DEFINITIONS cannot discard an alternate-key definition that has already been applied to the nested file. To delete an applied alternate-key definition, call AMP$DELETE_KEY_DEFINITION, and then call AMP$APPLY_KEY_DEFINITION to apply the deletion request.

AMP$APPLY_KEY_DEFINITIONS

Purpose Applies the pending alternate-key definitions and deletions to the file.

Format

AMP$APPLY_KEY_DEFINITIONS
(file_identifier, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$begin_altkey_labels
aae$begin_delete_keydefs
aae$duplicate_alternate_key
aae$enable_altkey_duplicates
aae$end_altkey_labels
aae$end_delete_keydefs
aae$index_being_built
aae$keydef_has_been_deleted
aae$no_definitions
aae$not_enough_permission
aae$sparse_key_beyond_eor
aae$unexpected_dup_encountered

Remarks

- AMP$APPLY_KEY_DEFINITIONS applies the pending requests to the currently selected nested file only. (The nested file selected when the file is opened is the default nested file, $MAIN_FILE.)

- An AMP$APPLY_KEY_DEFINITIONS call first deletes each alternate index specified by a pending alternate-key deletion. It then creates an alternate index for each pending alternate-key definition.

  A pending definition or deletion is one requested by an AMP$CREATE_KEY_DEFINITION or AMP$DELETE_KEY_DEFINTION call that has not yet been discarded or applied to the file. (Closing the file or issuing an AMP$ABANDON_KEY_DEFINITIONS call discards all pending definitions and deletions.)

Remarks (Contd)

- If AMC$NO_DUPLICATES_ALLOWED is specified for a new key and the file contains data, AMP$APPLY_KEY_DEFINITIONS returns a nonfatal error (condition AAE$UNEXPECTED_DUP_ENCOUNTERED) if it finds a duplicate alternate-key value. It then changes the duplicate control for the index from AMC$NO_DUPLICATES_ALLOWED to AMC$ORDERED_BY_PRIMARY_KEY, and restarts creation of the alternate index. (All other indexes are unaffected by this change.)

  If a change to AMC$ORDER_BY_PRIMARY_KEY is not desired, set the error_limit attribute to 1. The occurrence of a nonfatal error (such as a duplicate-key value) causes the nonfatal-error limit to be reached and a fatal error to be issued. The fatal error terminates alternate index creation and discards any alternate indexes already built by the call.

  No alternate indexes are created by the terminated AMP$APPLY_KEY_DEFINITIONS procedure; however, it does perform all pending alternate-key deletions.

- Entry of a pause_break_character (usually control-p) is ignored during application of alternate-key definitions.

- Entry of a terminate_break_character (usually control-t) during application of alternate-key definitions returns a prompt to the terminal user, asking for confirmation.

  As described in the prompt, the terminal user should then enter a carriage return or any entry other than RUIN FILE (uppercase or lowercase) to continue the application of alternate-key definitions. Applied alternate-key definitions can be removed without harm to the file after the apply operation has completed.

  A request to ruin the file is not recommended. No file operation can be performed on a ruined file and so no data can be retrieved from the file.

AMP$CREATE_KEY_DEFINITION

Purpose Defines an alternate key.

Format

AMP$CREATE_KEY_DEFINITION
(file_identifier, key_name, key_position, key_length, optional_attributes, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_name: amt$key_name

Name to be given the alternate key. The name must follow the SCL naming rules. It can be specified by an amt$key_name variable or by a 31-character string on the call. (The name must be left-justified with blank fill within the string.)

key_position: amt$key_position

Position of the first byte of the alternate key in the record. (The bytes in a record are numbered from the left, beginning with zero.)

key_length: amt$key_length

Length, in bytes, of the alternate key. The maximum length is 255 bytes.

optional_attributes: ^amt$optional_key_attributes

Pointer to an adaptable array defining optional attributes of the alternate key. Specify NIL if no optional attributes are to be specified.

Each record in the array specifies an optional attribute; the attribute defined is indicated by the SELECTOR field of the record. Table I-3-1 lists the SELECTOR field values and the attribute record fields generated for each SELECTOR field value.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$alt_key_past_minrl
aae$bad_name
aae$cant_create_existing_name
aae$concatenated_key_too_big
aae$cant_get_collate_table
aae$collated_altkey_no_table
aae$no_repeating_group

Remarks

- To apply the alternate-key definition specified by an AMP$CREATE_KEY_DEFINITION call to the file, call AMP$APPLY_KEY_DEFINITIONS. Before the apply operation, an alternate-key definition is only pending and cannot be used to access records in the file. A call to AMP$ABANDON_KEY_DEFINITIONS discards pending alternate-key definitions.

- If the SELECTOR field in a record in the optional_attributes array has the value AMC$NULL_ATTRIBUTE, that record is ignored.

- Sparse key control is defined by three values:

  Sparse_Key_Control_Position
  Sparse_Key_Control_Characters
  Sparse_Key_Control_Effect

  If an alternate key is subject to sparse-key control, the sparse-key control character must be within the minimum record length, but the alternate-key fields need not be. For more information, see the Sparse-Key Control description in chapter I-1.

- A concatenated key can have up to 64 pieces. The leftmost piece is defined by the key_position and key_length values.

  Each piece concatenated to the first piece is specified by a record in the optional_attributes array containing three fields:

  Concatenated_Key_Position
  Concatenated_Key_Length
  Concatenated_Key_Type

  The pieces are concatenated in the same order as the records that define the pieces in the optional_attributes array.

  The total length of a concatenated key cannot exceed 700 bytes.

- The first alternate key value in a repeating group begins at key_position. Subsequent keys are found by adding the value of repeating_group_length to key_position until either the repeating_group_count is satisfied (repeat_to_end_of_record is FALSE) or the end of the record is reached (repeat_to_end_of_record is TRUE).

Remarks (Contd)

- Repeating groups cannot be used with concatenated keys. Also, repeating groups cannot be used when duplicate_key_control is set to AMC$FIRST_IN_FIRST_OUT.

NOTE

The CYBIL declaration for AMT$OPTIONAL_KEY_ATTRIBUTE in appendix C lists additional fields besides those listed in table I-3-1. These additional fields are for features not yet implemented.

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE)

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC$KEY_TYPE	KEY_TYPE : amt$key_type Type of the alternate key. AMC$UNCOLLATED_KEY Order key values byte-by-byte according to the ASCII character set sequence (listed in appendix B). Key values can be positive integers or ASCII strings (1 through 255 bytes). AMC$INTEGER_KEY Order key values numerically. Key values are positive or negative integers (1 through 8 bytes). AMC$COLLATED_KEY Order key values according to a user-specified collation table (see the COLLATE_TABLE_NAME description in this table). Key values can be positive integers or ASCII strings (1 through 255 bytes). If you omit the attribute, AMC$UNCOLLATED_KEY is used.

(Continued)

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC$COLLATE_TABLE_NAME	COLLATE_TABLE_NAME : pmt$program_name Name of the collation table to be used for collating the alternate key. (The alternate-key collation table can differ from the primary-key collation table. See appendix D for more information on collation tables.) If the file is an indexed-sequential file with a collated primary key, the default collation table for the alternate key is the collation table for the primary key. Otherwise, you must specify a collation table for each collated alternate key.
AMC$DUPLICATE_KEYS	DUPLICATE_KEY_CONTROL : amt$duplicate_key_control Indicates how duplicate alternate-key values are handled in the alternate index. AMC$NO_DUPLICATES_ALLOWED No duplicate alternate-key values are allowed in the alternate index. AMC$FIRST_IN_FIRST_OUT Duplicate alternate-key values are ordered according to when the record is written to the file. AMC$ORDERED_BY_PRIMARY_KEY Duplicate alternate-key values are ordered according to primary-key values. Omission causes AMC$NO_DUPLICATES_ALLOWED to be used.

(Continued)

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC$NULL_ SUPPRESSION	NULL_SUPPRESSION : boolean Indicates whether alternate keys with a null value should be included in the alternate index. (For AMC$INTEGER_KEY, the null value is zero; for AMC$UNCOLLATED_ KEY, the null value is all spaces; for AMC$COLLATED_KEY, the null value is all spaces before collation.) FALSE All values are included in the index. TRUE Null values are not included in the index. Omission causes FALSE to be used.
AMC$SPARSE_KEYS	SPARSE_KEY_CONTROL_POSITION : amt$key_position Position of the sparse-key control character. The position must be within the minimum record length. (Bytes in a record are numbered from the left, beginning with zero.) SPARSE_KEY_CONTROL_ CHARACTERS : set of char Set of characters with which the sparse-key character is compared. SPARSE_KEY_CONTROL_EFFECT : amt$sparse_key_control_effect Indicates whether a sparse-key control character match causes the alternate key to be included or excluded from the alternate index.

(Continued)

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
	AMC$INCLUDE_KEY_VALUE Alternate-key value is included in the alternate index. AMC$EXCLUDE_KEY_VALUE Alternate-key value is not included in the alternate index.
AMC$REPEATING_GROUP	REPEATING_GROUP_LENGTH : amt$max_record_length, Length, in bytes, of the repeating group of fields. It is the distance from the beginning of an alternate-key value to the beginning of the next alternate-key value in the record. REPETITION_CONTROL : amt$repetition_control This record indicates whether the alternate key repeats until the end of the record. If no values are specified for the repetition_control record, it is assumed that the repeating group repeats until the end of the record. REPEAT_TO_END_OF_RECORD : boolean TRUE The alternate key repeats until the record ends. (An incomplete key at the end of the record is not used.) FALSE The alternate key repeats the number of times specified in the REPEATING_GROUP_COUNT field. If sparse-key control is not used, the specified number of key values must be within the minimum record length.

(Continued)

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
	REPEATING_GROUP_COUNT : amt$max_repeating_group_count Number of times the group of fields repeats in a record. This field exists only if REPEAT_TO_END_OF_RECORD is FALSE.
AMC$CONCATENATED_KEY_PORTION	CONCATENATED_KEY_POSITION : amt$key_position Starting position of a concatenated piece. (Bytes are numbered from the left, beginning with zero.) CONCATENATED_KEY_LENGTH : amt$key_length Length, in bytes, of a concatenated piece. CONCATENATED_KEY_TYPE : amt$key_type Key type of a concatenated piece. AMC$UNCOLLATED_KEY Order piece values byte-by-byte according to the ASCII character set sequence (listed in appendix B). Piece values can be positive integers or ASCII strings (1 through 255 bytes). AMC$INTEGER_KEY Order piece values numerically. Piece values are positive or negative integers (1 through 8 bytes). AMC$COLLATED_KEY Order piece values according to a user-specified collation table (see the COLLATE_TABLE_NAME description in this table). Piece values can be positive integers or ASCII strings (1 through 255 bytes).

AMP$CREATE_NESTED_FILE

Purpose Defines a nested file in an existing NOS/VE file.

Format

AMP$CREATE_NESTED_FILE
(file_identifier, definition, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

definition: amt$nested_file_definition

Variant record which specifies the nested-file name and its attributes. The record declaration is as follows:

```
amt$nested_file_definition = record
    nested_file_name: amt$nested_file_name,
    embedded_key: boolean,
    key_position: amt$key_position,
    key_length: amt$key_length,
    maximum_record: amt$max_record_length,
    minimum_record: amt$min_record_length,
    record_type: amt$record_type,
    case file_organization:
           amt$file_organization of
    = amc$indexed_sequential =
      key_type: amt$key_type,
      collate_table_name: pmt$program_name,
      data_padding: amt$data_padding,
      index_padding: amt$index_padding,
    = amc$direct_access =
      home_block_count:
        amt$initial_home_block_count,
      dynamic_home_block_space:
        amt$dynamic_home_block_space,
      loading_factor: amt$loading_factor,
      hashing_procedure:
        amt$hashing_procedure_name,
    casend,
recend;
```

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_name
aae$cant_get_collate_table
aae$collated_key_needs_table
aae$data_pad_too_large
aae$dup_nested_file_name
aae$index_pad_too_large
aae$integer_key_gt_one_word
aae$min_gt_max_rec_length
aae$no_home_block_count
aae$no_select_during_keydef
aae$not_enough_permission
aae$rec_too_small_for_key
aae$system_error_occurred

Remarks

- AMP$CREATE_NESTED_FILE requires append, modify, and shorten access to the file; otherwise, it returns condition aae$not_enough_permission.
- AMP$CREATE_NESTED_FILE cannot create a nested file if one or more alternate-key requests are pending. Call AMP$APPLY_KEY_DEFINITIONS or AMP$ABANDON_KEY_DEFINITIONS to dispose of the pending requests.
- The specified nested-file name must be unique among the nested files in the file; otherwise, AMP$CREATE_NESTED_FILE returns condition aae$dup_nested_file_name.
- You must specify values for all fields in the nested-file definition record that apply to the file organization. No default values are provided; the corresponding attribute values specified when the file was created apply only to the default nested file ($MAIN_FILE).

  The attributes and their values are described in chapter I-4.
- When creating an indexed-sequential nested file, specify OSC$NULL_NAME for the collate_table_name field when the key_type specified is AMC$UNCOLLATED or AMC$INTEGER. Specify the collation table name in the field when the key_type is AMC$COLLATED.
- If the key type is collated, specification of a collation table is required. AMP$CREATE_NESTED_FILE loads the collation table and stores it for use by the nested file. If it cannot load the collation table, it returns condition aae$cant_get_collate_table.

Remarks (Contd)

- When creating a direct-access nested file, specify values for the dynamic_home_block_space and loading_factor fields (although the values are not yet used). Specify the default values, FALSE and 0, respectively.

  For the hashing_procedure specification, values are required for two fields (NAME and OBJECT_LIBRARY). Currently, you should always specify OSC$NULL_NAME for the OBJECT_LIBRARY field. To specify the default hashing procedure, specify AMP$SYSTEM_HASHING_PROCEDURE as the NAME field value.

- Creating a nested file does not select the nested file for use. To select a nested file, call AMP$SELECT_NESTED_FILE.

- To remove a nested file, call AMP$DELETE_NESTED_FILE.

- For more information on nested files, see Nested Files in chapter I-1.

- The nested-file example at the end of chapter I-2 demonstrates the use of this call.

AMP$DELETE_KEY

Purpose Removes a record from a keyed file.

Format

AMP$DELETE_KEY
(file_identifier, key_location, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_location: ^cell

Pointer to the primary-key value of the record to be deleted.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$file_at_file_limit
aae$file_is_ruined
aae$key_not_already_locked
aae$key_not_found
aae$key_required
aae$nonembedded_key_not_given
aae$not_enough_permission

Remarks

- An AMP$DELETE_KEY call requires that the file be opened for at least shorten access. However, if the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate indexes can be updated.
- If the file could be shared (more than one instance of open could exist), a record can be deleted only by the owner of a Preserve_Access_and_Content or Exclusive_Access lock on the primary-key value of the record. An invalid attempt returns the nonfatal condition aae$key_not_already_locked.

  To read about file sharing, see Keyed-File Sharing in chapter I-2.
- When the delete request is executed, the specified record is either flagged as deleted or physically deleted from the data block. When the first record in a data block is deleted, index blocks are updated as applicable.

Remarks (Contd)

- If execution of a delete request empties a data or index block, the block is linked into a chain of empty blocks. These blocks are reused when new blocks are required for file expansion.

- AMP$DELETE_KEY searches for the specified primary-key value only in the nested file currently selected. If it does not find it, it returns the nonfatal condition aae$key_not_found.

- Execution of an AMP$DELETE_KEY call does not change the file position or the currently selected key.

  An AMP$DELETE_KEY call updates the alternate indexes if alternate keys are defined for the file. Calls to delete records are effective even if an alternate key is currently selected for reading and positioning the file.

- When deleting a series of contiguous fixed-length records, you can save execution time by beginning with the record having the highest primary-key value.

  Deletion of the last record in a data block is performed quickly because the system just needs to reduce the record count by one. Deletion of the first record in a data block, however, can move all remaining records in the data block.

  By deleting records in order from the highest to the lowest primary-key value, you can avoid relocation of records to be subsequently deleted.

AMP$DELETE_KEY_DEFINITION

Purpose Requests removal of an alternate-key definition by the next AMP$APPLY_KEY_DEFINITIONS call.

Format

AMP$DELETE_KEY_DEFINITION
(file_identifier, key_name, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_name: amt$key_name

Name of the alternate key to be deleted. It can be specified by an amt$key_name variable or by a 31-character string on the call. (The name must be left-justified with blank fill within the string.)

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$bad_name
aae$cant_delete_missing_name
aae$no_delete_current_key
aae$not_enough_permission

Remarks

- A subsequent AMP$APPLY_KEY_DEFINITIONS call is required to implement an alternate-key deletion specified by an AMP$DELETE_KEY_DEFINITION call.

  Before the apply operation, an alternate-key deletion is only pending; the alternate key remains in the file, although it is not available for use. (Another instance of open that has already selected the alternate key can continue to use it; however, no instance of open can select the key while its deletion is pending.)

  A call to AMP$ABANDON_KEY_DEFINITIONS discards pending alternate-key deletions.

- You cannot delete an alternate key while you have the key selected. Before calling AMP$DELETE_KEY_DEFINITION for the current key, you must call AMP$SELECT_KEY to select another key; otherwise AMP$DELETE_KEY_DEFINITION returns the condition aae$no_delete_current_key.

AMP$DELETE_NESTED_FILE

Purpose Destroys a nested file. It deletes its data, alternate keys, and the nested file definition.

Format

```
AMP$DELETE_NESTED_FILE
(file_identifier, nested_file_name, status);
```

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

nested_file_name: amt$nested_file_name

Name given the nested file when it was created. It can be specified by an amt$nested_file_name variable or by a 31-character string on the call. (The name must be left-justified with blank fill within the string.)

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_name
aae$cant_delete_main_nested_f
aae$nested_file_not_found
aae$no_delete_current_nested_f
aae$no_delete_rasp_in_use
aae$no_select_during_keydef
aae$not_enough_permission
aae$system_error_occurred

Remarks

- AMP$DELETE_NESTED_FILE requires append, modify, and shorten access to the file.
- The default nested file $MAIN_FILE cannot be deleted.
- The task must have exclusive access to the nested file to delete it. AMP$DELETE_NESTED_FILE cannot delete a nested file while:
  - Any instance of open has the nested file selected.
  - Any instance of open has any locks that apply to the nested file.

  An attempt to delete a nested file while it is in use returns the nonfatal condition aae$no_delete_rasp_in_use.

Remarks (Contd)

- AMP$DELETE_NESTED_FILE cannot delete a nested file if the instance of open has one or more alternate-key requests pending. Call AMP$APPLY_KEY_DEFINITIONS or AMP$ABANDON_KEY_DEFINITIONS to dispose of the pending requests and then call AMP$SELECT_NESTED_FILE to select another nested file.

  The default nested file $MAIN_FILE is the recommended selection while deleting nested files because $MAIN_FILE cannot be deleted.

- For more information on nested files, see Nested Files in chapter I-1.

- The nested-file example at the end of chapter I-2 demonstrates the use of this call.

AMP$GET_KEY

Purpose Reads a record from a keyed file using the specified key value.

Format

AMP$GET_KEY
(file_identifier, working_storage_area, working_storage_length, key_location, major_key_length, key_relation, record_length, file_position, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the space to which the record is copied.

working_storage_length: amt$working_storage_length

Length, in bytes, of the working storage area.

key_location: ^cell

Pointer to the key value of the record to be read. Set to NIL if the key value is an alternate-key value specified in the working storage area.

major_key_length: amt$major_key_length

Length of the major key in bytes. The major key length must be less than or equal to the key length.

If the value is zero, the full key length is used.

This parameter is ignored if the file is a direct-access file and its primary key is currently selected.

Parameters (Contd)

key_relation: amt$key_relation

Relationship between the key value of the record and the key value at key_location. The possible values are as follows:

AMC$EQUAL_KEY

AMP$GET_KEY reads the first record whose key value is equal to the key value at key_location.

AMC$GREATER_OR_EQUAL_KEY

AMP$GET_KEY reads the first record whose key value is equal to or greater than the key value at key_location.

AMC$GREATER_KEY

AMP$GET_KEY reads the first record whose key value is greater than the key value at key_location.

This parameter is ignored if the file is a direct-access file and its primary key is currently selected.

record_length: VAR of amt$max_record_length

Variable in which the number of bytes read is returned.

file_position: VAR of amt$file_position

Variable in which the file position at completion of the read operation is returned.

AMC$END_OF_KEY_LIST

File is positioned at the end of the key list for the alternate-key value specified on the call.

AMC$EOR

File is positioned at end-of-record.

AMC$EOI

File is positioned at end-of-information.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$file_at_file_limit
aae$file_is_ruined
aae$key_found_lock_no_wait
aae$key_not_found
aae$major_key_too_long
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$record_longer_than_wsa

Remarks

- To allow for updating of file statistics, you should open the file for both read and modify access.
- If the file could be shared (more than one concurrent instance of open could exist), the primary-key value of the record should be locked before the record is read. The program should either lock the key value before the AMP$GET_KEY call or replace the AMP$GET_KEY call with an AMP$GET_LOCK_KEYED_RECORD call.

  If another instance of open has an Exclusive_Access lock on the primary-key value of the record, AMP$GET_KEY returns the nonfatal condition aae$key_found_lock_no_wait and leaves the file positioned to read the record it found.

  To read about locks, see Keyed-File Sharing in chapter I-2.
- AMP$GET_KEY searches for the specified key value only in the currently selected nested file.
- AMP$GET_KEY can read a record by its primary-key value or by an alternate-key value. The primary key is used unless a preceding AMP$SELECT_KEY call has selected an alternate key.
- If the primary key is selected, the key_location parameter must point to the location of the key value.
- If an alternate key is selected, the key_location parameter can point to the location of the key or it can be set to NIL.

  If key_location is set to NIL, AMP$GET_KEY expects the key to be in the working storage area. The location of the key in the working storage area must match the location of the key in the record.

  If the alternate key is a concatenated key, each field in the concatenated key must be stored in its appropriate location in the working storage area.

Remarks (Contd)

- For an indexed-sequential file, AMP$GET_KEY uses a major key if the major_key_length parameter value is nonzero. A major key is the leftmost bytes of the key. AMP$GET_KEY searches for the lowest key value that begins with the major-key value or, if that is not found, a value greater than the major-key value.
- For an indexed-sequential file, the nonfatal condition aae$key_not_found is returned if no record in the nested file has a key value that satisfies the relation specified by the key_relation parameter (equal, greater than, or greater than or equal). AMP$GET_KEY always positions the file at the point where the record satisfying the relation would be located if it existed in the file.
- AMP$GET_KEY returns the actual length of the record in the variable specified by the record_length parameter. If the length of the record is greater than the length of the working storage area, AMP$GET_KEY returns working_storage_length characters to the working storage area; it also returns the nonfatal condition aae$record_longer_than_wsa.
- File positioning by AMP$GET_KEY differs depending on the file organization and the key selected.
- For a direct-access file with its primary key selected, the following statements are true:
  - An AMP$GET_KEY call does not change the file position used by sequential access calls.
  - The only file_position value AMP$GET_KEY returns is AMC$EOR.
  - The only calls that reposition the file are the AMP$REWIND call and the sequential access calls (AMPGET_NEXT, AMPGET_NEXT_KEY, AMP$GET_LOCK_NEXT_KEY).
  - The major_key_length and key_relation parameter values are not used.
- An AMP$GET_KEY call for a direct-access file with an alternate key selected is processed the same as a call to an indexed-sequential file with an alternate key selected.

Remarks (Contd)

- For an indexed-sequential file, execution of the AMP$GET_KEY call leaves the file positioned at the end of the record that was read. (AMC$EOR or AMC$END_OF_KEY_LIST is returned in the file_position parameter.)

  When AMP$GET_KEY returns AMC$EOI as the file position, it has not found the requested record and does not return data in the working storage area. It returns AMC$EOI in both of these cases:

  - It is searching for a key value that is greater than or equal to the specified key value and the specified key value is greater than all key values in the file.
  - It is searching for a key value that is greater than the specified key value and the specified key value is the highest value in the file.

AMP$GET_KEY_DEFINITIONS

Purpose Retrieves the definitions of all alternate keys in the file.

Format

AMP$GET_KEY_DEFINITIONS
(file_identifier, key_definitions, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_definitions: SEQ(*)

Sequence to receive the description of the alternate keys.

Each definition is written in two parts: a record of type AMT$BASIC_KEY_DEFINITION and an array of type AMT$OPTIONAL_KEY_ATTRIBUTES records containing four or more additional records. (The number of records is returned in the NUMBER_OF_OPTIONAL_ATTRIBUTES field of the AMT$BASIC_KEY_DEFINITION record.)

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$not_enough_permission
aae$too_little_space

Remarks

- A successful AMP$GET_KEY_DEFINITIONS call returns a sequence of key definitions. The last key definition in the sequence consists of an AMT$BASIC_KEY_DEFINITION record in which the field DEFINITION_RETURNED is FALSE; the record serves as the terminator for the sequence of key definitions.
- If the DEFINITION_RETURNED field is TRUE in an AMT$BASIC_KEY_DEFINITION record, the record is the first part of a key definition. The NUMBER_OF_OPTIONAL_ATTRIBUTES field in the record specifies the number of additional records returned for the key definition; the records are returned in an array of type AMT$OPTIONAL_KEY_ATTRIBUTES.

Remarks (Contd)

- The SELECTOR field of an optional attribute record indicates the attribute returned in the record. The possible attributes are: key_type, duplicate_key_control, null_suppression, group_name, sparse_key_control, concatenated_key, and repeating_groups. The first four records are returned for every key definition; the subsequent records are returned only if the attribute was specified for the key definition.

- The attribute order in a key definition may not match the attribute order specified when the alternate key was defined. However, the returned definition is logically equivalent and, if used to redefine the key, results in an identical alternate key.

- All name values in an alternate-key definition are returned using uppercase letters only (even if lowercase letters were used when the name was originally specified).

Example

The following CYBIL statements show how the key definition sequence returned by an AMP$GET_KEY_DEFINITIONS call could be read. The key definition sequence is declared to be 500 words long (500 integers). If the sequence is too small, AMP$GET_KEY_DEFINITIONS returns the condition AAE$TOO_LITTLE_SPACE.

Example (Contd)

```
MODULE GET_DEFS_MOD;
*copyc amp$open
*copyc amp$get_key_definitions
*copyc amt$optional_key_attributes
PROCEDURE GET_ALT_KEY_DEFS;
  VAR
    lfn: [STATIC] amt$local_file_name :=
                  'existing_is_file',
    fid: amt$file_identifier,
    status: ost$status,
    definitions_ptr : SEQ (*),
    definitions : SEQ(REP 500 OF integer),
    basic_definition : amt$basic_key_definition,
    optional_attributes : amt$optional_key_attributes;

  amp$open(lfn,amc$record,NIL,fid,status);

{ Statements here to check the status variable.}

  amp$get_key_definitions (fid,definitions,status);

{ Statements here to check the status variable.}

  definitions_ptr := definitions;
  RESET definitions_ptr;

{ Set the basic_definitions pointer to the first record.}
  NEXT basic_definition IN definitions_ptr;

{ Iterate until the definition_returned field in the }
{ basic_definition record is FALSE.}
  WHILE basic_definition.definition_returned DO

{ Set the optional_attributes pointer to the beginning }
{ of the optional attributes array.}
  NEXT optional_attributes :
  [1 .. basic_definition.number_of_optional_attributes]
     IN definitions_ptr;
              { : }
{ Use the key definition here. }
              { : }
{ Set the basic_definition pointer to the next key }
{ definition.}
  NEXT basic_definition IN definitions_ptr;
  WHILEND;
PROCEND GET_ALT_KEY_DEFS;
MODEND GET_DEFS_MOD
```

AMP$GET_LOCK_KEYED_RECORD

Purpose Locks and reads the record having the specified key value.

Format

AMP$GET_LOCK_KEYED_RECORD (file_identifier, working_storage_area, working_storage_length, key_location, major_key_length, key_relation, wait_for_lock, unlock_control, lock_intent, record_length, file_position, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the space to which the record is copied.

working_storage_length: amt$working_storage_length

Length, in bytes of the working storage area.

key_location: ^cell

Pointer to the key value of the record to be read. Set to NIL if the key value is an alternate-key value specified in the working storage area.

major_key_length: amt$major_key_length

Length of the major key in bytes. The major key length must be less than or equal to the key length.

If the major key length is zero, the full key length is used.

This parameter is ignored if the file is a direct-access file and its primary key is currently selected.

Parameters (Contd)

key_relation: amt$key_relation

Relationship between the key value of the record and the key value specified by this call. The valid values are as follows:

AMC$EQUAL_KEY	Read the first record whose key value is equal to the specified key value.
AMC$GREATER_OR_EQUAL_KEY	Read the first record whose key value is greater than or equal to the specified key value.
AMC$GREATER_KEY	Read the first record whose key value is greater than the specified key value.

This parameter is ignored if the file is a direct-access file and its primary key is currently selected.

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns immediately with a warning condition if the lock is unavailable.

unlock_control: amt$unlock_control

Indicates whether the lock is to be cleared automatically.

AMC$AUTOMATIC	Clear the lock automatically.
AMC$WAIT_FOR_UNLOCK	Keep the lock until it is explicitly unlocked.

lock_intent: amt$lock_intent

Specifies the purpose and effects of the lock.

AMC$EXCLUSIVE_ACCESS	Locked for exclusive access.
AMC$PRESERVE_ACCESS_AND_CONTENT	Locked for possible update request later.
AMC$PRESERVE_CONTENT	Locked to read the record only.

Parameters (Contd)

record_length: VAR of amt$max_record_length

Variable in which the number of bytes read is returned.

file_position: VAR of amt$file_position

Variable at which the file position at completion of the read operation is returned.

AMC$END_OF_KEY_LIST	Positioned at the end of the key list for the specified alternate-key value.
AMC$EOR	Positioned at the end of the record.
AMC$EOI	Positioned at the end-of-information.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$file_at_file_limit
aae$file_is_ruined
aae$key_already_locked
aae$key_deadlock
aae$key_expired_lock_exists
aae$key_found_lock_no_wait
aae$key_not_found
aae$key_self_deadlock
aae$key_timeout
aae$lock_file_crowded
aae$major_key_too_long
aae$no_auto_unlock_pc
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$primary_key_locked
aae$record_longer_than_wsa
aae$too_many_keylocks

Remarks

- To allow for updating of file statistics, you should open the file for both read and modify access.
- AMP$GET_LOCK_KEYED_RECORD performs the same processing as AMP$GET_KEY except that it locks the primary-key value of the record before reading the record. See the AMP$GET_KEY procedure description for details on how AMP$GET_LOCK_KEYED_RECORD finds and reads the record.
- AMP$GET_LOCK_KEYED_RECORD requests a lock on the primary-key value of the record to be read. The lock request uses the wait_for_lock, unlock_control, and lock_intent values on the call. For more information on locks, see Keyed-File Sharing in chapter I-2.
- Because a preserve_content lock cannot be automatically unlocked, the unlock_control value AMC$AUTOMATIC and the lock_intent value AMC$PRESERVE_CONTENT are not valid on the same call.
- If an alternate key is currently selected, the call requests a lock on the first primary-key value in the key list only.
- If the call terminates abnormally, the primary-key value is left unlocked.
- If the requested lock is unavailable, the call leaves the file positioned to read the requested record.

AMP$GET_LOCK_NEXT_KEYED_RECORD

Purpose Locks and reads the next record.

Format

AMP$GET_LOCK_NEXT_KEYED_RECORD (file_identifier, working_storage_area, working_storage_length, key_location, wait_for_lock, unlock_control, lock_intent, record_length, file_position, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the space to which the record is copied.

working_storage_length: amt$working_storage_length

Length, in bytes of the working storage area.

key_location: ^cell

Pointer to the space in which the key value of the record is returned.

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns a warning condition if the lock is unavailable.

unlock_control: amt$unlock_control

Indicates whether the lock is to be cleared automatically.

AMC$AUTOMATIC	Clear the lock automatically.
AMC$WAIT_FOR_UNLOCK	Keep the lock until it is explicitly unlocked.

Parameters (Contd)

lock_intent: amt$lock_intent

Specifies the purpose and effects of the lock.

AMC$EXCLUSIVE_ ACCESS	Locked for exclusive access.
AMC$PRESERVE_ ACCESS_AND_ CONTENT	Locked for possible update request later.
AMC$PRESERVE_ CONTENT	Locked to read the record only.

record_length: VAR of amt$max_record_length

Variable in which the number of bytes read is returned.

file_position: VAR of amt$file_position

Variable at which the file position at completion of the read operation is returned.

AMC$END_OF_KEY_ LIST	Positioned at the end of the key list for the specified alternate-key value.
AMC$EOR	Positioned at the end of the record.
AMC$EOI	Positioned at the end-of-information.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$cant_da_getn_if_shared
aae$cant_da_getn_after_put
aae$cant_position_beyond_bound
aae$file_at_file_limit
aae$file_boundary_encountered
aae$file_is_ruined
aae$key_already_locked
aae$key_deadlock
aae$key_expired_lock_exists
aae$key_found_lock_no_wait
aae$key_self_deadlock
aae$key_timeout
aae$lock_file_crowded
aae$no_auto_unlock_pc
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$primary_key_locked
aae$record_longer_than_wsa
aae$too_many_keylocks
aae$wsa_not_given

Remarks

- To allow for updating of file statistics, you should open the file for both read and modify access.
- AMP$GET_LOCK_NEXT_KEYED_RECORD performs the same processing as AMP$GET_NEXT_KEY except that it locks the primary-key value of the record before reading the record. See the AMP$GET_NEXT_KEY procedure description for details on how AMP$GET_LOCK_NEXT_KEYED_RECORD finds and reads the record.
- AMP$GET_LOCK_NEXT_KEYED_RECORD requests a lock on the primary-key value of the record to be read. The lock request uses the wait_for_lock, unlock_control, and lock_intent values on the call. For more information on locks, see Keyed-File Sharing in chapter I-2.
- Because a Preserve_Content lock cannot be automatically unlocked, the unlock_control value AMC$AUTOMATIC and the lock_intent value AMC$PRESERVE_CONTENT are not valid on the same call.
- If an alternate key is currently selected, the call requests a lock on the first primary-key value in the key list only.
- If the call terminates abnormally, the primary-key value is left unlocked.

Remarks (Contd)

- If the requested lock is unavailable, the call leaves the file positioned to read the requested record.
- This call is valid for a direct-access file only when an alternate key is selected or during a sequential pass through the file.

  When the primary key is selected, the call is valid only when the direct-access file has been attached for exclusive access (no share modes allowed) and no update operations intervene in the sequential pass. (The only update operation allowed is the replacement of a record with another record of the same length.)

  If an update operation is performed on the direct-access file and the primary key is selected, the program must rewind the file before beginning a sequential pass of the direct-access file.

AMP$GET_NESTED_FILE_DEFINITIONS

Purpose Returns the nested-file definitions for the file.

Format

```
AMP$GET_NESTED_FILE_DEFINITIONS
(file_identifier, definitions, nested_file_count, status);
```

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

definitions: VAR of amt$nested_file_definitions

Array in which the nested-file definitions are returned. Each element is a record of type amt$nested_file_definition as described for the AMP$CREATE_NESTED_FILE procedure.

nested_file_count: VAR of amt$nested_file_count

Variable in which the number of nested files in the file is returned.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$too_little_space
aae$not_enough_permission
aae$system_error_occurred

Remarks

- AMP$GET_NESTED_FILE_DEFINITIONS requires the same access required to open the file.
- The definition of the currently selected nested file is always returned first in the nested_file_definitions array.
- If the nested_file_definitions array is too small for all nested-file definitions in the file, AMP$GET_NESTED_FILE_DEFINITIONS returns the nonfatal condition aae$too_little_space.

  In this case, if sufficient space is available, it returns the definition of the currently selected nested file in the first element of the array, but leaves the rest of the array undefined.

  After receiving the condition aae$too_little_space, a program can use the nested_file_count returned to increase the size of the array to that required for all nested-file definitions and then call AMP$GET_NESTED_FILE_DEFINITIONS again.

Remarks (Contd)

- To fetch the number of nested files before calling AMP$GET_NESTED_FILE_DEFINITIONS, call AMP$FETCH_ACCESS_INFORMATION to fetch the amc$number_of_nested_files item. (AMP$FETCH_ACCESS_INFORMATION is described in the CYBIL File Management manual.)
- All name values in a nested-file definition are returned using uppercase letters only (even if lowercase letters were used when the name was originally specified).
- Besides using the individual field values returned in the nested-file definition record, you can use the records returned to create similar or identical nested files in another file. This can be done easily because the record type returned by AMP$GET_NESTED_FILE_DEFINITIONS is the same record type specified on an AMP$CREATE_NESTED_FILE call.
- For more information on nested files, see Nested Files in chapter I-1.
- The nested-file example at the end of chapter I-2 demonstrates the use of this call.

AMP$GET_NEXT_KEY

Purpose Reads the next logical record in the keyed file.

Format

```
AMP$GET_NEXT_KEY
(file_identifier, working_storage_area, working_
storage_length, key_location, record_length, file_
position, wait, status);
```

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the space to which the record is copied.

working_storage_length: amt$working_storage_length

Length, in bytes, of the working storage area.

key_location: ^cell

Pointer to the space in which the record key value is returned.

record_length: VAR of amt$max_record_length

Variable in which the number of bytes read is returned.

file_position: VAR of amt$file_position

Variable in which the position of the file at completion of the read operation is returned.

AMC$END_OF_KEY_LIST

File is positioned at the end of a key list (can be returned only if an alternate key was selected).

AMC$EOR

File is positioned at the end of a record. (When an alternate key is selected, it indicates that the file is not at the end of a key list.)

AMC$EOI

File is positioned at the end of the index.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$cant_position_beyond_bound
aae$file_at_file_limit
aae$file_boundary_encountered
aae$file_is_ruined
aae$key_found_lock_no_wait
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$record_longer_than_wsa
aae$wsa_not_given

Remarks

- When a file is being read but not updated, the file should be opened for both read and modify access. The modify access allows statistics to be updated without allowing any record in the file to be altered.

- If the file could be shared (more than one concurrent instance of open could exist), the primary-key value of the record should be locked before the record is read. Either a call before the AMP$GET_NEXT_KEY call should lock the key value or an AMP$GET_LOCK_NEXT_KEYED_RECORD call should replace the AMP$GET_NEXT_KEY call.

  If another instance of open has an Exclusive_Access lock on the primary-key value of the record, AMP$GET_NEXT_KEY returns the nonfatal condition aae$key_found_lock_no_wait and leaves the file positioned to read the record it found.

  To read about locks, see Keyed-File Sharing in chapter I-2.

- AMP$GET_NEXT_KEY reads the next record in the currently selected nested file.

- When an alternate key is selected, get_next calls return records in the key-value order as provided by the alternate index.

  When the primary key is selected for an indexed-sequential file, records are returned in the key-value order as provided by the primary index.

  When the primary key is selected for a direct-access file, records are not returned in a logical order; records are returned in physical order by their location in the file.

Remarks (Contd)

- AMP$GET_NEXT_KEY returns the file_position AMC$EOR (or AMC$END_OF_KEY_LIST for an alternate key) when it returns a record to the working storage area.

  When AMP$GET_NEXT_KEY reads the last record in the file, it returns AMC$EOR (or AMC$END_OF_KEY_LIST for an alternate key) as the file position. The next AMP$GET_NEXT_KEY call returns AMC$EOI as the file position; it returns no data and normal status. If the task calls AMP$GET_NEXT_KEY again after AMC$EOI has been returned, the status condition AAE$CANT_POSITION_BEYOND_BOUND occurs.

  For more information on the use of this call with alternate keys, refer to Using Alternate Keys in chapter I-2.

- The key value is returned to key_location unless the key_location parameter is set to NIL.

- At the completion of the read request, the record_length parameter is set to the length of the record that was read. If the sequential read operation was unsuccessful, the record_length parameter is not defined.

- If the length of the record that is read is greater than the length of the working storage area as specified by the working_storage_length parameter, working_storage_length characters are returned and a nonfatal error occurs.

- This call is valid for a direct-access file only when an alternate key is selected or during a sequential pass through the file.

  When the primary key is selected, the call is valid only when the direct-access file has been attached for exclusive access (no share modes allowed) and no update operations intervene in the sequential pass. (The only update operation allowed is the replacement of a record with another record of the same length.)

  If an update operation is performed on the direct-access file and the primary key is selected, the program must rewind the file before beginning a sequential pass of the direct-access file.

AMP$GET_NEXT_PRIMARY_KEY_LIST

Purpose Returns a list of primary-key values associated with a range of alternate-key values in an alternate index.

Format

AMP$GET_NEXT_PRIMARY_KEY_LIST (file_identifier, high_key, major_high_key, high_key_relation, working_storage_area, working_storage_length, end_of_primary_key_list, transferred_byte_count, transferred_key_count, file_position, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

high_key: ^cell

Pointer to the alternate-key value at which the range ends. Set to NIL if the range ends at the end of the alternate index.

major_high_key: amt$major_key_length

Specify a nonzero value to indicate that the upperbound alternate-key value is to be located by major key. The nonzero value is the number of characters beginning at the high_key location that are to be used as the major key. Specify zero to indicate that the full alternate-key value is to be used.

high_key_relation: amt$key_relation

Indicates where the list ends in relation to the highest alternate-key value in the range.

AMC$GREATER_KEY

Include the primary-key values associated with the high_key value in the list; that is, end the list when an alternate-key value greater than the high_key value is encountered.

AMC$GREATER_OR_EQUAL_KEY or AMC$EQUAL_KEY

Exclude the primary-key values associated with the high_key value from the list; that is, end the list when an alternate-key value greater than or equal to the high_key value is encountered.

working_storage_area: ^cell

Pointer to the variable in which the list of primary-key values is returned.

Parameters (Contd)

working_storage_length: amt$working_storage_length

Length, in bytes, of the working storage area.

end_of_primary_key_list: VAR of boolean

Variable in which a boolean value is returned indicating whether the entire list of primary-key values was returned to the working storage area.

TRUE

The high end of the range was reached, and the entire list of primary-key values was returned to the working storage area.

FALSE

The high end of the range was not reached, and at least one more AMP$GET_NEXT_PRIMARY_KEY_LIST call is required to get the rest of the list of primary-key values.

transferred_byte_count: VAR of amt$working_storage_length

Variable in which the length, in bytes, of the list of primary-key values is returned.

transferred_key_count: VAR of amt$key_count_limit

Variable in which the number of primary-key values is returned.

file_position: VAR of amt$file_position

Variable in which the file position at completion of the operation is returned.

AMC$EOR

File is positioned within a key list.

AMC$END_OF_KEY_LIST

File is positioned at the end of a key list.

AMC$EOI

File is positioned at the end of the alternate index.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$high_end_below_current
aae$not_enough_permission
aae$not_positioned_by_altkey
aae$wsa_not_given
aae$wsl_too_short

Remarks

- You must call AMP$SELECT_KEY to select the alternate key before calling AMP$GET_NEXT_PRIMARY_KEY_LIST; otherwise, AMP$GET_NEXT_PRIMARY_KEY_LIST returns the nonfatal error aae$not_positioned_by_altkey and does not return a list of primary-key values.

- The high_key parameter points to a value that specifies the upper bound of the range of keys to be listed. The high_key_relation parameter indicates whether the high_key value is included or excluded from the range.

  For example, suppose the high_key value is SMITH. The high_key_relation value indicates whether the primary-key values associated with the alternate-key value SMITH is included in the list.

- A major key consists of the leftmost bytes of a key. If the major_high_key parameter value is nonzero, AMP$GET_NEXT_PRIMARY_KEY_LIST uses a major key of the specified length to find the high end of the range. It searches for the lowest alternate-key value that begins with the major-key or a value greater than the major key.

  For example, suppose the key at the specified high_key location is ABCDEF. If the major_high_key parameter value is 2, the major key used is AB. Therefore, the range ends at the first alternate-key value beginning with AB.

- If high_key is set to NIL, the values of major_high_key and high_key_relation are ignored.

- A primary-key value can be included more than once in the list returned by AMP$GET_NEXT_PRIMARY_KEY_LIST. This occurs if the primary-key value is associated with more than one alternate-key value in the range. This is possible if the repeating-groups attribute is defined for the alternate key.

Remarks (Contd)

- AMP$GET_NEXT_PRIMARY_KEY_LIST returns primary-key values until it reaches the end of the specified range or until it cannot fit another value into the working storage area. By checking the end_of_primary_key_list value, the program can determine whether all requested values were returned and, if not, call AMP$GET_NEXT_PRIMARY_KEY_LIST again to fetch the rest of the values.

- AMP$GET_NEXT_PRIMARY_KEY_LIST repositions the file as it fetches key values. At completion of the call, the file is positioned at the end of the last key value returned and positioned to continue fetching values at that point if AMP$GET_NEXT_PRIMARY_KEY_LIST is called again.

AMP$GET_PRIMARY_KEY_COUNT

Purpose Returns the number of primary-key values associated with a range of alternate-key values in an alternate index.

Format

AMP$GET_PRIMARY_KEY_COUNT (file_identifier, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, list_count_limit, list_count, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

low_key: ^cell

Pointer to the alternate-key value at which the range begins. Set to NIL if the range is to begin at the lowest alternate-key value in the alternate index.

major_low_key: amt$major_key_length

A nonzero value indicates that the lowerbound alternate-key value is to be located by major key. The nonzero value is the major-key length. A zero value indicates that the full alternate-key value is to be used.

low_key_relation: amt$key_relation

Indicates where the count begins in relation to the lowest value in the alternate-key range.

AMC$GREATER_KEY

Exclude the primary keys associated with the low_key value from the count, that is, begin the count when an alternate-key value greater than the low_key value is encountered.

AMC$GREATER_OR_EQUAL_KEY or AMC$EQUAL_KEY

Include the primary keys associated with the low_key value in the count, that is, begin the count when an alternate-key value greater than or equal to the low_key value is encountered.

high_key: ^cell

Pointer to the alternate-key value at which the range ends. Set to NIL if the range ends at the highest alternate-key value in the alternate index.

Parameters (Contd)

major_high_key: amt$major_key_length

A nonzero value indicates that the upperbound alternate-key value is to be located by major key. The nonzero value is the major-key length. A zero value indicates that the full alternate-key value is to be used.

high_key_relation: amt$key_relation

Indicates where the count ends in relation to the highest value in the range.

AMC$GREATER_KEY

Include the primary-key values associated with the high_key value in the count; that is, end the count when an alternate-key value greater than the high_key value is encountered.

AMC$GREATER_OR_EQUAL_KEY or AMC$EQUAL_KEY

Exclude the primary-key values associated with the high_key value from the count; that is, end the count when an alternate-key value greater than or equal to the high_key value is encountered.

list_count_limit: amt$key_count_limit

Maximum number of primary-key values counted; AMP$GET_PRIMARY_KEY_COUNT stops counting when it reaches this value. If set to zero, all primary-key values are counted.

list_count: VAR of amt$key_count_limit

Integer variable in which the number of primary-key values in the range is returned. If zero is returned, no primary-key values exist in the specified range. The value cannot exceed the list count limit.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$high_end_not_above_low_end
aae$not_enough_permission
aae$not_positioned_by_altkey

Remarks

- You must call AMP$SELECT_KEY to select the alternate key before calling AMP$GET_PRIMARY_KEY_COUNT; otherwise, AMP$GET_PRIMARY_KEY_COUNT returns the nonfatal error aae$not_positioned_by_altkey and does not return a primary-key count.

- The low_key and high_key parameters point to values that specify the lower and upper bounds, respectively, of the alternate-key range. The low_key_relation and high_key_relation parameters indicate whether the low_key and high_key values, respectively, are included in the range.

  For example, suppose the low_key value is JONES and the high_key value is SMITH. The low_key_relation value indicates whether the primary keys associated with alternate-key value JONES are included in the count. The high_key_relation value indicates whether the primary keys associated with alternate-key value SMITH are included in the count.

- A major key consists of the leftmost characters of a key. The major_high_key and major_low_key parameters specify the number of characters of the specified key to use when searching for a matching key. A key is considered to match the specified key when the major key matches the first characters of the key.

  For example, suppose the key at the specified low_key position is ABCDEF. If the major_low_key parameter value is 2, the major key used is AB. Therefore, the count begins at the first alternate-key value beginning with a value greater than or equal to AB.

- If low_key is set to NIL, the values of major_low_key and low_key_relation are ignored. If high_key is set to NIL, the values of major_high_key and high_key_relation are ignored.

- AMP$GET_PRIMARY_KEY_COUNT counts a single primary-key value more than once if the primary-key value is associated with more than one alternate-key value. This is possible if the repeating groups attribute is defined for the alternate key.

Remarks (Contd)

- AMP$GET_PRIMARY_KEY_COUNT returns the value 0 as the list count if it cannot find both the upper_bound and lower_bound alternate-key values in the alternate index.

  For example, if you specify the alternate-key value Z as both the upper_bound and the lower_bound values and the alternate-key value Z is not in the alternate index, the call returns 0 as the list count.

- The list_count_limit value can minimize the processing required for the call. For example, if you call AMP$GET_PRIMARY_KEY_COUNT call to determine whether the number of primary-key values for the alternate-key value Z is 0, 1, or more than 1, you should set the list_count_limit value to 2.

AMP$GET_SPACE_USED_FOR_KEY

Purpose Returns the number of alternate-index blocks that contain the specified alternate-key range.

Format

AMP$GET_SPACE_USED_FOR_KEY (file_identifier, low_key, major_low_key, low_key_ relation, high_key, major_high_key, high_key_ relation, data_block_count, data_block_space, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

low_key: ^cell

Pointer to the alternate-key value at which the range begins. Set to NIL if the range is to begin at the lowest alternate-key value in the alternate index.

major_low_key: amt$major_key_length

A nonzero value indicates that the lowerbound is specified by a major key (the leftmost part of the key). The nonzero value is the major key length. A zero value indicates that the full alternate-key value is to be used.

low_key_relation: amt$key_relation

Indicates where the count begins in relation to the lowest value in the alternate-key range.

AMC$GREATER_KEY

Exclude the low_key value from the range.

AMC$GREATER_OR_EQUAL_KEY or AMC$EQUAL_KEY

Include the low_key value in the range.

high_key: ^cell

Pointer to the alternate-key value at which the range ends. Set to NIL if the range ends at the highest alternate-key value in the alternate index.

major_high_key: amt$major_key_length

A nonzero value indicates that the upperbound is specified by a major key (the leftmost part of the key). The nonzero value is the major key length. A zero value indicates that the full alternate-key value is to be used.

Parameters (Contd)

high_key_relation: amt$key_relation

Indicates where the range ends in relation to the highest value in the range.

AMC$GREATER_KEY

Include the high_key value in the range.

AMC$GREATER_OR_EQUAL_KEY or AMC$EQUAL_KEY

Exclude the high_key value from the range.

data_block_count: VAR of amt$data_block_count

Variable in which the block count is returned. It is returned as an integer from 1 through amt$max_blocks_per_file.

data_block_space: VAR of amt$file_length

Variable in which the combined length of the blocks is returned. (The value is the number of blocks multiplied by the block size.)

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$high_end_not_above_low_end
aae$not_enough_permission
aae$not_positioned_by_altkey

Remarks

- The structure of an alternate index is an indexed-sequential structure. One or more index levels are used to find the block containing the alternate-key value. Only the blocks at the lowest level of the search actually contain the alternate-key values and their corresponding primary-key values.

  An AMP$GET_SPACE_USED_FOR_KEY call does not actually find the specified alternate-key values in the alternate index. Rather, it searches the index to determine the number of lowest-level blocks that would contain the specified range of alternate-key values.

  AMP$GET_SPACE_USED_FOR_KEY returns a value even if the specified low_key and high_key values are not in the alternate index.

Remarks (Contd)

- This call can be used to compare alternate methods of fetching a set of primary-key values. This is discussed under Retrieving Alternate-Index Information in chapter I-2.
- An AMP$GET_SPACE_USED_FOR_KEY call returns two values, a block count and the combined length of the blocks counted. The second value is derived by multiplying the block count by the block size for the file. It is useful when comparing values from files with different block sizes. (Larger blocks require longer searches.)
- An alternate key must be currently selected when AMP$GET_SPACE_USED_FOR_KEY is called. If the primary key is currently selected, AMP$GET_SPACE_USED_FOR_KEY returns the nonfatal error aae$not_positioned_by_altkey and does not return block_count or block_length values.
- The low_key, major_low_key, low_key_relation, high_key, major_high_key, and high_key_relation parameters specify the range of alternate-key values. Their use on an AMP$GET_SPACE_USED_FOR_KEY call is the same as on an AMP$GET_PRIMARY_KEY_COUNT call. For details, see the Remarks in the AMP$GET_PRIMARY_KEY_COUNT description.

AMP$LOCK_FILE

Purpose Locks the file.

Format

AMP$LOCK_FILE
(file_identifier, wait_for_lock, lock_intent, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns immediately with a warning condition if the lock is unavailable.

lock_intent: amt$lock_intent

Specifies the purpose and effects of the lock.

AMC$EXCLUSIVE_ACCESS	Locked for exclusive access.
AMC$PRESERVE_ACCESS_AND_CONTENT	Locked for possible update requests later.
AMC$PRESERVE_CONTENT	Locked to read records only.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$key_timeout
aae$lock_file_crowded

Remarks

- The file lock applies to the currently selected nested file only. It applies to all primary-key values in that nested file and to all lock requests for the nested file.
- File locks are not automatically unlocked. A file lock is cleared when one of these events occurs:
  - An AMP$UNLOCK_FILE call clears the lock.
  - The instance of open is closed.
- For more information, see File Locks in chapter I-2.

AMP$LOCK_KEY

Purpose Locks the specified primary-key value.

Format

AMP$LOCK_KEY
(file_identifier, key_location, wait_for_lock, unlock_control, lock_intent, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_location: ^cell

Pointer to the primary-key value to be locked.

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns immediately with a warning condition if the lock is unavailable.

unlock_control: amt$unlock_control

Indicates whether the lock is automatically cleared.

AMC$AUTOMATIC	The lock is cleared by the next request that reads, updates, or positions the file or requests or clears a lock.
AMC$WAIT_FOR_UNLOCK	The lock is held until it is explicitly cleared.

AMC$AUTOMATIC is not valid if the lock_intent value is AMC$PRESERVE_CONTENT.

Parameters (Contd)

lock_intent: amt$lock_intent

Specifies the purpose and effects of the lock.

AMC$EXCLUSIVE_ ACCESS	Locked for exclusive access.
AMC$PRESERVE_ ACCESS_AND_ CONTENT	Locked for possible update request later.
AMC$PRESERVE_ CONTENT	Locked to read the record only.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$key_already_locked
aae$key_deadlock
aae$key_expired_lock_exists
aae$key_found_lock_no_wait
aae$key_self_deadlock
aae$key_timeout
aae$lock_file_crowded
aae$no_auto_unlock_pc
aae$primary_key_locked
aae$too_many_keylocks

Remarks

- Only primary-key values can be locked; alternate-key values cannot be locked. The currently selected key does not affect AMP$LOCK_KEY.
- The key lock applies only to the nested file currently selected.
- The specified primary-key value may or may not be that of a record in the nested file.
  - If the primary-key value is already in the nested file, the lock prevents access to the record associated with that primary-key value.
  - If the primary-key value is not yet in the nested file, the lock reserves the key value for a record to be written by the task. No other task can write a record with that primary-key value while the lock is in effect.

Remarks (Contd)

- AMP$LOCK_KEY does not verify that the primary-key value is valid. The validity of the key value is determined by a subsequent call that uses the key value.
- Because a Preserve_Content lock cannot be automatically unlocked, the unlock_control value AMC$AUTOMATIC and the lock_intent value AMC$PRESERVE_CONTENT are not valid on the same call.
- If automatic unlock is not chosen for the key lock, the lock is not cleared until one of these events occurs:
  - An AMP$UNLOCK_KEY call clears the lock.
  - The instance of open is closed.
- For more information, see Keyed-File Sharing in chapter I-2.

AMP$PUT_KEY

Purpose Writes a record to a keyed file.

Format

AMP$PUT_KEY
(file_identifier, working_storage_area, working_storage_length, key_location, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the new record.

working_storage_length: amt$working_storage_length

Length, in bytes, of the record to be written.

key_location: ^cell

Pointer to the primary-key value of the new record; specify NIL if the primary key is embedded.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$duplicate_alternate_key
aae$file_at_file_limit
aae$file_at_user_record_limit
aae$file_full_no_puts_or_reps
aae$file_is_ruined
aae$key_already_exists
aae$key_found_lock_no_wait
aae$key_required
aae$nonembedded_key_not_given
aae$not_enough_permission

Remarks

- An AMP$PUT_KEY call requires that the file be opened for at least append access. If the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate indexes can be updated.

- A lock is not required for an AMP$PUT_KEY call. However, if the file could be shared (more than one concurrent instance of open could exist), the primary-key value of the record should be locked before the record is written. A Preserve_Content_and_Access or Exclusive_Access lock prevents another task from writing a record with the same primary-key value.

  If another instance of open has a lock on the primary-key value, AMP$PUT_KEY returns the nonfatal condition aae$key_found_lock_no_wait.

  To read about file sharing, see Keyed-File Sharing in chapter I-2.

- AMP$PUT_KEY writes the record in the nested file currently selected.

- If the primary key is nonembedded, the key_location parameter specifies the starting address of the key. If the primary key is embedded, the key_location parameter is ignored, and the location of the key is determined by the key_position attribute; therefore, you should specify the key_location parameter as NIL.

- If the file has AMC$ANSI_FIXED records, the working_storage_length parameter is ignored, and the value of the max_record_length attribute is used as the length of the working storage area.

  A warning message is issued for the first call on which the working_storage_length value differs from the max_record_length value. The warning is given because, although excess data is truncated, insufficient data in the working storage area is not padded. This could mean that garbage has been written as the last part of the fixed-length record.

- Execution of an AMP$PUT_KEY call does not change the key currently selected. It leaves the file positioned at the end of the record it writes.

Remarks (Contd)

- Writing records to an indexed-sequential file is usually faster if the records are sorted in ascending primary-key value order before being written to the file. Also, the resulting file is usually smaller.

  Writing unsorted records to an indexed-sequential file could result in an inefficient file structure with more data blocks than necessary because of numerous data-block splits.

- An AMP$PUT_KEY call updates the alternate indexes for the new record if alternate keys are defined for the file. Calls to put or replace records are effective even if an alternate key is currently selected for reading and positioning the file.

- AMP$PUT_KEY returns one of these nonfatal conditions when it cannot write the record because the nested file has reached a limit:

  `aae$file_at_user_record_limit`
  The number of records in the nested file has reached the record_limit attribute value.

  `aae$file_full_no_puts_or_reps`
  The record cannot be written because it would require addition of another index level to the indexed-sequential structure and the number of index levels has already reached the limit (15).

AMP$PUTREP

Purpose Either replaces a record if the record is in the keyed file or adds a new record if the record is not in the file.

Format

AMP$PUTREP
(file_identifier, working_storage_area, working_storage_length, key_location, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the new record.

working_storage_length: amt$working_storage_length

Length, in bytes, of the record to be written.

key_location: ^cell

Pointer to the primary-key value of the new record; specify NIL if the primary key is embedded.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$file_at_file_limit
aae$file_at_user_record_limit
aae$file_full_no_puts_or_reps
aae$file_is_ruined
aae$key_found_lock_no_wait
aae$key_required
aae$nonembedded_key_not_given

Remarks

- An AMP$PUTREP call requires that the file be opened with at least append and shorten access. If the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate indexes can be updated.
- AMP$PUTREP writes or replaces a record in the nested file currently selected.

Remarks (Contd)

- If the primary-key value specified on the call matches the primary-key value of a record in the nested file, AMP$PUTREP performs the same processing as an AMP$REPLACE_KEY call.

  If the primary-key value specified on the call does not match any primary-key value in the nested file, AMP$PUTREP performs the same processing as an AMP$PUT_KEY call.

  The only exception to the preceding statements is that, even if the primary-key values match, an AMP$PUTREP call does not require a lock on the specified primary-key value (unlike an AMP$REPLACE_KEY call which requires a lock if the file is shared).

- If the file could be shared (more than one concurrent instance of open could exist), the primary-key value of the record should be locked before the record is written or replaced. A Preserve_Content_and_Access or Exclusive_Access lock prevents another task from writing or replacing the record.

  If another instance of open has a lock on the primary-key value, AMP$PUTREP returns the nonfatal condition aae$key_found_lock_no_wait.

  To read about file sharing, see Keyed-File Sharing in chapter I-2.

AMP$REPLACE_KEY

Purpose

Replaces an existing record in a keyed file with a new record having the same primary-key value.

Format

AMP$REPLACE_KEY
(file_identifier, working_storage_area, working_storage_length, key_location, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

working_storage_area: ^cell

Pointer to the new record.

working_storage_length: amt$working_storage_length

Length, in bytes, of the record to be written.

key_location: ^cell

Pointer to the primary-key value of the new record; specify NIL if the primary key is embedded.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$duplicate_alternate_key
aae$file_at_file_limit
aae$file_full_no_puts_or_reps
aae$file_is_ruined
aae$key_not_found
aae$key_required
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$sparse_key_beyond_eor

Remarks

- An AMP$REPLACE_KEY call requires that the file be opened with at least append and shorten access. If the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate index can be updated.

Remarks (Contd)

- If the file could be shared (more than one instance of open could exist), a record can be replaced only by the owner of a Preserve_Access_and_Content or Exclusive_Access lock on the primary-key value of the record. An invalid attempt returns the nonfatal condition aae$key_not_already_locked.

  To read about file sharing, see Keyed-File Sharing in chapter I-2.

- AMP$REPLACE_KEY searches for the specified primary-key value only in the nested file currently selected. If it does not find it, it returns the nonfatal condition aae$key_not_found.

- The replace request fails if the file does not contain a record whose primary-key value matches the primary-key value of the replacement record. It returns the nonfatal condition aae$key_not_found; file processing can continue.

- If the record type of the file is AMC$VARIABLE or AMC$UNDEFINED, the new record can be shorter or longer than the existing record; however, the length of the new record must be within the minimum and maximum record length values defined for the file.

- For AMC$ANSI_FIXED type records, the value of working_storage_length is ignored and the fixed record length (the max_record_length attribute value) is used.

  A warning message is issued for the first call on which the working_storage_length value differs from the max_record_length value. The warning is given because, although excess data is truncated, insufficient data in the working storage area is not padded. This could mean that garbage has been written as the last part of the fixed-length record.

- Execution of an AMP$REPLACE_KEY call does not change the file position or the currently selected key.

  An AMP$REPLACE_KEY call updates the alternate indexes for the new record if alternate keys are defined for the file. Calls to put or replace records are effective even if an alternate key is currently selected for reading and positioning the file.

AMP$SELECT_KEY

Purpose Selects the key to be used by subsequent calls that read or position the file.

Format

```
AMP$SELECT_KEY
(file_identifier, key_name, status);
```

Parameters

file_identifier: amt$file_identifier
File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_name: amt$key_name
Name of the key to be used. It can be specified by an amt$key_name variable or by a 31-character string on the call. (The name must be left-justified with blank fill within the string.)

Specify the name $PRIMARY_KEY to switch from an alternate key back to the primary key.

status: VAR of ost$status
Status variable in which the completion status is returned.

Condition Identifiers

aae$altkey_name_not_found
aae$cant_select_key
aae$cant_select_until_applied
aae$no_select_on_pending_delete
aae$not_enough_permission

Remarks

- The initial key selected when a file is opened is always the primary key.
- The key selection remains in effect until another AMP$SELECT_KEY call is issued or the file is closed.
- AMP$SELECT_KEY cannot select an alternate key for which a deletion request is pending (an AMP$DELETE_KEY_DEFINITION call has specified the key). If a deletion request is pending for the specified key, AMP$SELECT_KEY returns the condition aae$no_select_on_pending_delete.
- When an AMP$SELECT_KEY call changes the selected key, it positions the file at the record having the lowest key value for the selected key (that is, it rewinds the file for that key). However, if the AMP$SELECT_KEY call does not change the selected key (the key specified on the call is already selected), it does not rewind the file (the file is left in its current position).

AMP$SELECT_NESTED_FILE

Purpose Selects a nested file for use.

Format

AMP$SELECT_NESTED_FILE
(file_identifier, nested_file_name, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

nested_file_name: amt$nested_file_name

Name given the nested file when it was created or $MAIN_FILE, the name of the default nested file.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$no_select_during_keydef
aae$nested_file_not_found
aae$cant_select_nested_file
aae$not_enough_permission
aae$system_error_occurred

Remarks

- AMP$SELECT_NESTED_FILE requires the same access required to open the file.
- An Exclusive_Access file lock prevents other instances of open from selecting the nested file. AMP$SELECT_NESTED_FILE returns the nonfatal condition aae$cant_select_nested_file.
- The default nested file ($MAIN_FILE) is initially selected when the file is opened.
- All requests specifying the file identifier apply to the selected nested file until another AMP$SELECT_NESTED_FILE call selects another nested file.
- The initial file position of each nested file is the open_position of the file. The initially selected key is $PRIMARY_KEY.

Remarks (Contd)

- AMP$SELECT_NESTED_FILE does not discard the file position, selected key, or locks of previously selected nested files. The instance of open keeps this information for all nested files.

  Thus, a task can sequentially access records on one nested file, select another nested file, reselect the first nested file, and continue the sequential access.

  Similarly, when a task selects an alternate key and then selects another nested file, the alternate key remains selected for the first nested file.

- AMP$SELECT_NESTED_FILE cannot select another nested file if one or more alternate key requests are pending. Call AMP$APPLY_KEY_DEFINITIONS or AMP$ABANDON_KEY_DEFINITIONS to dispose of the pending requests.

- To fetch the name of the currently selected nested file, call AMP$FETCH_ACCESS_INFORMATION to fetch the amc$selected_nested_file item. (AMP$FETCH_ACCESS_INFORMATION is described in the CYBIL File Management manual.)

- For more information on nested files, see Nested Files in chapter I-1.

AMP$START

Purpose Positions the file to the beginning of the first record in the file having a key value that satisfies the specified key relation. A record is not returned to the working storage area.

Format

AMP$START
(file_identifier, key_location, major_key_length, key_relation, file_position, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_location: ^cell

Location of the key value to which the key value of each record in the file is compared.

major_key_length: amt$major_key_length

Length of the major key in bytes. The major key is the leftmost bytes of the key at key_location. The major key is compared to the leftmost bytes of a key.

If the value is zero, a full-length key is used to position the file. Otherwise, the number of bytes specified for the major_key_length parameter must be less than or equal to the value of the key_length attribute.

key_relation: amt$key_relation

Relationship between the key of the record and the key at key_location. The possible values are as follows:

AMC$EQUAL_KEY

The key value of the record equals the key value at key_location.

AMC$GREATER_OR_EQUAL_KEY

The key value of the record equals the key value at key location or, if an equal key value does not exist, it is the next greater key value.

AMC$GREATER_KEY

The key value of the record is the first key value greater than the key value at key_location.

Parameters (Contd)

file_position: VAR amt$file_position

File position at completion of the start operation.

AMC$END_OF_KEY_LIST

File is positioned to read the first record containing the alternate-key value specified on the call (that is, at the end of the preceding key list, if one exists).

AMC$EOR

File is positioned to access the record containing the primary-key value specified on the call (that is, at the end of the preceding record, if one exists).

AMC$EOI

File is positioned at the end-of-information.

wait: ost$wait

Currently, the only valid value is OSC$WAIT. You must specify this value on the call.

status: VAR ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$file_at_file_limit
aae$file_is_ruined
aae$key_not_found
aae$major_key_too_long
aae$no_da_or_sk_start
aae$nonembedded_key_not_given
aae$not_enough_permission

Remarks

- An AMP$START call requires that the file be opened for at least read access.
- AMP$START searches for the specified key value in the nested file currently selected.
- The current file position does not affect AMP$START processing.
- For direct-access files, an AMP$START call is valid only if an alternate key is currently selected. If the primary key is selected, an AMP$START call for a direct-access file returns the nonfatal condition aae$no_da_or_sk_start.

Remarks (Contd)

- The AMP$START call does not specify a working storage area, so the key value cannot be specified in the working storage area as it can on other calls. Instead, the key_location parameter must point to the location of the key value.

- If an alternate key has been selected and the key is a concatenated key, the values for the key fields must be assembled at key_location. The key fields must be concatenated in order as defined for the key.

  For example, if the key is the last three bytes of the record followed by the first three bytes of the record, the value at key_location must be the last three bytes followed by the first three bytes. For more information on concatenated keys, see the description in chapter I-1.

- If no key value in the file satisfies the specified key_relation with the specified key value, AMP$START returns the nonfatal condition aae$key_not_found. The file is left positioned either at the beginning of the first record whose key value is greater than the specified key value or, if the specified key value is greater than all key values in the file, at the end-of-information.

- A lock on a primary-key value does not prevent AMP$START from positioning the file using that key value.

  Like other file request calls, an AMP$START call clears any lock requested with automatic unlock.

AMP$UNLOCK_FILE

Purpose Clears a file lock.

Format

```
AMP$UNLOCK_FILE
(file_identifier, status);
```

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Remarks

- An AMP$UNLOCK_FILE call clears the file lock for the currently selected nested file only.

  To clear all file locks and all key locks belonging to the instance of open, call AMP$UNLOCK_KEY and specify TRUE for the unlock_all_keys parameter.

- When a lock expires, the task must clear the lock before it can perform any other operations on any nested file in the file.

- For more information, see Keyed-File Sharing in chapter I-2.

AMP$UNLOCK_KEY

Purpose Clears locks.

Format

AMP$UNLOCK_KEY
(file_identifier, unlock_all_keys, key_location, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

unlock_all_keys: boolean

Indicates whether the call clears all locks or only a single key lock.

TRUE	Clears all locks for the instance of open.
FALSE	Clears only the key lock for the value at key_location.

key_location: ^cell

Pointer to the primary-key value to be unlocked. The value is ignored if unlock_all_keys is true.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$key_already_locked
aae$key_deadlock
aae$key_expired_lock_exists
aae$key_found_lock_no_wait
aae$key_self_deadlock
aae$key_timeout
aae$lock_file_crowded
aae$no_auto_unlock_pc
aae$primary_key_locked
aae$too_many_keylocks

Remarks

- AMP$UNLOCK_KEY performs one of two operations depending on the value of the unlock_all_keys parameter:
  - Clears all locks belonging to the instance of open. This includes all file locks and all key locks for all nested files.
  - Clears only the key lock for the primary-key value specified at key_location. The key lock must apply to the currently selected nested file.
- AMP$UNLOCK_KEY cannot clear an individual nested-file lock. To do so, call AMP$UNLOCK_FILE.
- If the call is to unlock all locks, but no locks exists for the instance of open, the call does nothing and returns normal status. However, if the call is to clear a single key lock and the lock does not exist, the call returns the nonfatal condition aae$key_not_previously_locked.
- When a lock expires, the task must clear the lock before it can perform any other operations on any nested file in the file. (A lock can expire only if the lock_expiration_time attribute for the file is not zero.)

  The task is not notified as to which lock has expired. The most direct response to a lock expiration condition is to call AMP$UNLOCK_KEY to clear all locks.

Keyed-File Attributes I-4

Keyed-File Attribute and Access Item Descriptions I-4-5
ACCESS_MODE I-4-5
AVERAGE_RECORD_LENGTH I-4-8
COLLATE_TABLE I-4-9
COLLATE_TABLE_NAME I-4-10
DATA_PADDING I-4-11
DUPLICATE_VALUE_INSERTED I-4-12
EMBEDDED_KEY I-4-12
EOI_BYTE_ADDRESS I-4-13
ERROR_COUNT I-4-13
ERROR_EXIT_NAME I-4-14
ERROR_EXIT_PROCEDURE I-4-15
ERROR_LIMIT I-4-15
ERROR_STATUS I-4-16
ESTIMATED_RECORD_COUNT I-4-16
FILE_LENGTH I-4-16
FILE_LIMIT I-4-17
FILE_ORGANIZATION I-4-17
FILE_POSITION I-4-18
FORCED_WRITE I-4-19
GLOBAL_ACCESS_MODE I-4-20
GLOBAL_FILE_NAME I-4-20
GLOBAL_SHARE_MODE I-4-21
HASHING_PROCEDURE_NAME I-4-21
INDEX_LEVELS I-4-22
INDEX_PADDING I-4-22
INITIAL_HOME_BLOCK_COUNT I-4-23
KEY_LENGTH I-4-23
KEY_POSITION I-4-23
KEY_TYPE I-4-24
LAST_ACCESS_OPERATION I-4-25
LAST_OP_STATUS I-4-27
LEVELS_OF_INDEXING I-4-27
LOCK_EXPIRATION_TIME I-4-27
MAX_BLOCK_LENGTH I-4-28
MAX_RECORD_LENGTH I-4-28
MESSAGE_CONTROL I-4-29
MIN_RECORD_LENGTH I-4-30
NULL_ATTRIBUTE I-4-30
NULL_ITEM I-4-30
NUMBER_OF_NESTED_FILES I-4-31
OPEN_POSITION I-4-31
PERMANENT_FILE I-4-32
PRIMARY_KEY I-4-32

RECORD_LIMIT I-4-32
RECORD_TYPE I-4-33
RECORDS_PER_BLOCK I-4-33
RESIDUAL_SKIP_COUNT I-4-34
RETURN_OPTION I-4-34
RING_ATTRIBUTES I-4-35
SELECTED_KEY_NAME I-4-35
SELECTED_NESTED_FILE I-4-36

Keyed-File Attributes I-4

Like all other NOS/VE files, a keyed file has a set of file attributes. The CYBIL procedure calls to specify and fetch attribute values are described in detail in the CYBIL File Management manual. This chapter describes the file attributes applicable to keyed files.

Besides file attribute values, a CYBIL program can also fetch file access information items that pertain only to a specific instance of open. (It fetches file access information using the AMP$FETCH_ACCESS_INFORMATION call described in the CYBIL File Management manual.)

This chapter describes the file access information items applicable to keyed files. If you request an item for a keyed file that does not apply to keyed files (and thus, is not described in this chapter), AMP$FETCH_ACCESS_INFORMATION does not return a value for the item. To indicate this, it sets the boolean ITEM_RETURNED field in the item record to FALSE.

Table I-4-1 lists the keyed-file attributes and access information items and the calls that can specify or fetch the values.

Table I-4-1. Keyed-File Attributes and Access Information Items

FETCH = AMP$FETCH
FETCH_INFO = AMP$FETCH_ACCESS_INFORMATION
FILE = AMP$FILE
GET = AMP$GET_FILE_ATTRIBUTES
OPEN = AMP$OPEN
STORE = AMP$STORE

Attribute or Item	FETCH	FETCH_ INFO	FILE	GET	OPEN	STORE
Access_Mode	X		X	X	X	
Average_Record_ Length	X		X	X	X	
Collate_Table	X					
Collate_Table_ Name	X		X	X	X	
Data_Padding	X		X	X	X	
Duplicate_Value_ Inserted		X				
Embedded_Key	X		X	X	X	
EOI_Byte_Address		X				
Error_Count		X				
Error_Exit_Name	X		X	X	X	
Error_Exit_ Procedure	X					X
Error_Limit	X		X	X	X	X
Error_Status		X				
Estimated_ Record_Count	X		X	X	X	
File_Length				X		
File_Limit	X		X	X	X	
File_Organization	X		X	X	X	
File_Position		X				
Forced_Write	X		X	X	X	
Global_Access_ Mode	X			X		
Global_File_ Name	X			X		

(Continued)

Table I-4-1. Keyed-File Attributes and Access Information Items *(Continued)*

FETCH = AMP$FETCH
FETCH_INFO = AMP$FETCH_ACCESS_INFORMATION
FILE = AMP$FILE
GET = AMP$GET_FILE_ATTRIBUTES
OPEN = AMP$OPEN
STORE = AMP$STORE

Attribute or Item	FETCH	FETCH_ INFO	FILE	GET	OPEN	STORE
Global_Share_ Mode	X			X		
Hashing_ Procedure_Name	X		X	X	X	
Index_Levels	X		X	X	X	
Index_Padding	X		X	X	X	
Initial_Home_ Block_Count	X		X	X	X	
Key_Length	X		X	X	X	
Key_Position	X		X	X	X	
Key_Type	X		X	X	X	
Last_Access_ Operation		X				
Last_Op_Status		X				
Levels_Of_ Indexing		X				
Lock_Expiration_ Time	X		X	X	X	
Max_Block_ Length	X		X	X	X	
Max_Record_ Length	X		X	X	X	
Message_Control	X		X	X	X	X

(Continued)

Table I-4-1. Keyed-File Attributes and Access Information Items *(Continued)*

FETCH = AMP$FETCH
FETCH_INFO = AMP$FETCH_ACCESS_INFORMATION
FILE = AMP$FILE
GET = AMP$GET_FILE_ATTRIBUTES
OPEN = AMP$OPEN
STORE = AMP$STORE

Attribute or Item	FETCH	FETCH_ INFO	FILE	GET	OPEN	STORE
Min_Record_ Length	X		X	X	X	
Null_Attribute	X		X	X	X	X
Null_Item		X				
Number_Of_ Nested_Files		X				
Open_Position	X		X	X	X	
Permanent_File	X			X		
Primary_Key		X				
Record_Limit	X		X	X	X	
Record_Type	X		X	X	X	
Records_Per_ Block	X		X	X	X	
Residual_ Skip_Count		X				
Return_Option			X	X	X	
Ring_Attributes	X		X	X	X	
Selected_ Key_Name		X				
Selected_ Nested_File		X				

Keyed-File Attribute and Access Item Descriptions

Each of the following attribute descriptions provides the following information:

- Name. (The name given is the name of the value field in the record specifying or fetching the value; the attribute or item identifier is the name with the prefix AMC$. For example, the identifier for ACCESS_MODE is AMC$ACCESS_MODE.)
- Meaning of the attribute or item for keyed-file use.
- Valid values.
- Default value for preserved and temporary attributes.
- The calls that can specify or fetch the attribute or access information item.

The descriptions follow in alphabetical order.

ACCESS_MODE

Meaning Set of access modes allowed during the instance of open (temporary attribute).

The access mode set limits the valid file operations during the instance of open. (The access modes required for each keyed-file interface call are listed in table I-4-2.)

Value Set of access mode identifiers (specified using the set identifier PFTUSAGE_SELECTIONS [ ]).

PFC$READ	Read access.
PFC$SHORTEN	Shorten access.
PFC$APPEND	Append access.
PFC$MODIFY	Modify access.
PFC$EXECUTE	Execute access.

The set can contain only access modes included in the global_access_mode set (see the global_access_mode attribute description).

Default Value The set of access modes defined by the global_access_mode attribute excluding PFC$EXECUTE.

The attribute cannot be changed during the instance of open.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

Table I-4-2. Required Access Modes for Calls

Call	Access Modes Required
AMP$ABANDON_KEY_DEFINITIONS	Append, shorten, and modify
AMP$APPLY_KEY_DEFINITIONS	Append, shorten, and modify
AMP$CREATE_KEY_DEFINITION	Append, shorten, and modify
AMP$CREATE_NESTED_FILE	Append, shorten, and modify
AMP$DELETE_KEY	Shorten
AMP$DELETE_KEY_DEFINITION	Append, shorten, and modify
AMP$DELETE_NESTED_FILE	Append, shorten, and modify
AMP$GET_KEY	Read (modify required to record statistics)
AMP$GET_KEY_DEFINITIONS	Any access mode
AMP$GET_LOCK_KEYED_RECORD	Read (modify required to record statistics; shorten or append required for an Exclusive_Access lock)
AMP$GET_LOCK_NEXT_KEYED_RECORD	Read (modify required to record statistics; shorten or append required for an Exclusive_Access lock)
AMP$GET_NESTED_FILE_DEFINITIONS	Any access mode
AMP$GET_NEXT_KEY	Read (modify required to record statistics)
AMP$GET_NEXT_PRIMARY_KEY_LIST	Read

(Continued)

Table I-4-2. Required Access Modes for Calls *(Continued)*

Call	Access Modes Required
AMP$GET_PRIMARY_KEY_COUNT	Read
AMP$GET_SPACE_USED_FOR_KEY	Read
AMP$LOCK_FILE	Any access mode (shorten or append required for an Exclusive_Access lock)
AMP$LOCK_KEY	Any access mode (shorten or append required for an Exclusive_Access lock)
AMP$PUT_KEY	Append (read, shorten, or modify also required if the file is not positioned at its EOI)
AMP$PUTREP	Append and shorten
AMP$REPLACE_KEY	Append and shorten
AMP$SELECT_NESTED_FILE	Any access mode
AMP$SELECT_KEY	Any access mode
AMP$START	Read
AMP$UNLOCK_FILE	Any access mode
AMP$UNLOCK_KEY	Any access mode

AVERAGE_RECORD_LENGTH

Meaning Estimate of the average record length in bytes (preserved attribute). If specified, the system uses the attribute value to calculate the block size used; it uses the attribute value only when opening a new file.

For ANSI fixed-length (F) records, the average_record_length value should be the same as the max_record_length value.

For variable (V) and undefined (U) records, the average_record_ length value depends on whether the majority of the records are the same length.

- If almost all records are a specific length, set the attribute value to that length.
- If the record lengths are well distributed within a range of lengths, set the attribute value to the median record length (half of the records are longer, half are shorter).

Value Integer from 1 through AMC$MAXIMUM_RECORD (type AMT$AVERAGE_RECORD_LENGTH).

Default Value None. If no value is set for the attribute, the system uses the arithmetic mean of the max_record_length and min_record_length values to calculate block size. Although the system uses that value, it does not store the value as the average_record_length value.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

COLLATE_TABLE

Meaning Collation table (returned attribute). This attribute is used to fetch the collation table stored with a file.

NOTE

To fetch the collation table, you specify a pointer in the COLLATE_TABLE field of the attribute record for an AMP$FETCH call. AMP$FETCH copies the collation table to the variable to which the pointer points. If you do not specify a pointer, the system attempts to use an undefined pointer and returns an error.

Value Pointer of type ^AMT$COLLATE_TABLE. Type AMT$COLLATE_TABLE has the following declaration:

```
ARRAY [CHAR] OF AMT$COLLATION_VALUE
```

Type AMT$COLLATION_VALUE is the integer subrange 0 through 255.

To determine the collating weight the table assigns to a particular character code, you use the character as the index into the table; the value at that position is the collating weight of that character.

For example, assume an AMP$FETCH call has fetched the collation table of a file and stored it in a variable named COLLATION_TABLE. The following statement assigns the collating weight of A to integer variable A_WEIGHT:

```
A_WEIGHT := COLLATION_TABLE['A'];
```

Assume the statement assigns the value 0 to A_WEIGHT. This means that the collation table assigns the collating weight 0 to character A.

Calls AMP$FETCH.

COLLATE_TABLE_NAME

Meaning Collation table name (preserved attribute). This attribute is used to specify a collation table for a file.

The attribute value is used only when the file is first opened. When the file is opened, the named collation table is stored with the file. The collation table for a file cannot be changed after a new file has been first opened.

Value 31-character program name (PMT$PROGRAM_NAME).

NOTE

All letters in the name must be specified as uppercase letters.

The name can be that of a system-defined collation table or a user-defined collation table. Collation table definition is described in appendix D, Collation Tables.

The names of the system-defined collation tables follow. The collating sequence for each table is listed in appendix D.

OSV$ASCII6_FOLDED
CYBER 170 FORTRAN 5 default collating sequence; lowercase letters mapped to uppercase letters.

OSV$ASCII6_STRICT
CYBER 170 FORTRAN 5 default collating sequence.

OSV$COBOL6_FOLDED
CYBER 170 COBOL 5 default collating sequence; lowercase letters mapped to uppercase letters.

OSV$COBOL6_STRICT
CYBER 170 COBOL 5 default collating sequence.

OSV$DISPLAY63_FOLDED
CYBER 170 63-character display code collating sequence; lowercase letters mapped to uppercase letters.

OSV$DISPLAY63_STRICT
CYBER 170 63-character display code collating sequence.

OSV$DISPLAY64_FOLDED
CYBER 170 64-character display code collating sequence; lowercase letters mapped to uppercase letters.

OSV$DISPLAY64_STRICT
CYBER 170 64-character display code collating sequence.

OSV$EBCDIC
Full EBCDIC collation sequence.

OSV$EBCDIC6_FOLDED
EBCDIC 6-bit subset supported by CYBER 170 COBOL 5 and SORT 5; lowercase letters mapped to uppercase letters.

OSV$EBCDIC6_STRICT
EBCDIC 6-bit subset supported by CYBER 170 COBOL 5 and SORT 5.

Default Value None. You must specify a value for the collate_table_name attribute if you specify AMC$INDEXED_SEQUENTIAL or AMC$DIRECT_ACCESS as the file_organization attribute value and AMC$COLLATED_KEY as the key_type attribute value.

If a collation table is stored with an indexed-sequential file, it becomes the default collation table for any collated alternate keys defined for the file.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

DATA_PADDING

Meaning Percentage of space the system is to leave empty in each data block created during the first instance of open of an indexed-sequential file (preserved attribute). The empty space allows for easy file expansion by later file processing operations.

The attribute value is used only during the first instance of open of an indexed-sequential file.

Value 0 through 99 (type AMT$DATA_PADDING).

Default Value 0 (no padding).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

DUPLICATE_VALUE_INSERTED

Meaning Indicates whether the last AMPPUT, AMPPUTREP, AMP$REPLACE, or AMP$APPLY_KEY_DEFINIIONS call detected a duplicate alternate-key value (access information item).

The duplicate_value_inserted item does not identify the duplication. An AMPPUT, AMPPUTREP, or AMP$REPLACE call can detect a duplicate value for any alternate key in the file that allows duplicates. An AMP$APPLY_KEY_DEFINITIONS call can detect a duplicate value for any record in the file.

Value Boolean value.

TRUE	The last call detected a duplicate alternate-key value.
FALSE	The last call did not detect a duplicate alternate-key value.

Calls AMP$FETCH_ACCESS_INFORMATION.

EMBEDDED_KEY

Meaning Indicates whether the primary key is part of the record data (preserved attribute).

Value Boolean value.

TRUE	The primary key is part of the record data.
FALSE	The primary key is separate from the record data.

Default Value TRUE.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

EOI_BYTE_ADDRESS

Meaning Current length of the file in bytes (access information item).

Value Integer from 0 through AMC$FILE_BYTE_LIMIT (type AMT$FILE_BYTE_ADDRESS).

Calls AMP$FETCH_ACCESS_INFORMATION.

ERROR_COUNT

Meaning Number of nonfatal (trivial) errors returned by keyed-file access requests (access information item).

Value Integer from 0 through AMC$MAX_ERROR_COUNT (type AMT$ERROR_COUNT).

Calls AMP$FETCH_ACCESS_INFORMATION.

ERROR_EXIT_NAME

Meaning Name of an error processing procedure (temporary attribute).

The name must be that of a procedure with the XDCL attribute within the program library list of the job or defined within the task.

For the attribute to be effective, you must specify the error_exit_name value before the file is opened or on the AMP$OPEN call. The error processing procedure is loaded when the file is opened. To change the procedure while the file is open, you must use the error_exit_procedure attribute.

Value 1- through 31-character procedure name (type PMT$PROGRAM_ NAME). (All letters in the name must be uppercase because PMP$LOAD does not convert lowercase letters to uppercase.)

The named procedure must be of type AMT$ERROR_EXIT_PROCEDURE; that is, it must have the following parameter list:

```
(file_identifier: AMT$FILE_IDENTIFIER;
 VAR status: OST$STATUS)
```

Default Value None. If no error-exit name is specified, the system does not search for an error-processing procedure.

For more information, see the error-exit procedure discussion in the CYBIL File Management Manual.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

ERROR_EXIT_PROCEDURE

Meaning Pointer to the current error processing procedure (temporary attribute).

You use this attribute to change the effective error processing procedure while the file is open. To clear the effective error processing procedure, specify a NIL pointer for the attribute.

Value Pointer variable of this type:

```
^procedure(file_identifier: amt$file_identifier;
           VAR status: ost$status)
```

Default Value None. The system continues to use the error processing procedure specified by the error_exit_name attribute when the file was opened, if one was specified.

For more information, see the error-exit procedure discussion in the CYBIL File Management Manual.

Calls AMP$FETCH, AMP$STORE.

ERROR_LIMIT

Meaning Maximum number of nonfatal (trivial) errors that can occur before the nonfatal errors cause a fatal error (temporary attribute).

A nonfatal error is an error that prevents successful completion of the current request, but does not prevent processing of subsequent requests. Its error severity level is ERROR.

Value Integer from 0 through 0FFFF(16) (type AMT$ERROR_LIMIT). 0 means no error limit.

Default Value 0 (no error limit).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN, AMP$STORE.

ERROR_STATUS

Meaning Completion status returned by the last file interface request for the file (access information item).

Value Integer (type OST$STATUS_CONDITION).

Calls AMP$FETCH_ACCESS_INFORMATION.

ESTIMATED_RECORD_COUNT

Meaning Estimated number of records to be stored in the file (preserved attribute).

The system uses the attribute value to calculate the block size; it only uses the value when it first opens a new file.

Value Integer (type AMT$ESTIMATED_RECORD_COUNT).

Default Value If a value is defined for the record_limit attribute, the record_limit value is the default estimated_record_count. If the record_limit attribute is undefined, the default value is 100,000.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

FILE_LENGTH

Meaning Length of a mass storage file in bytes (returned attribute).

Value Integer from 0 through AMC$FILE_BYTE_LIMIT, 4398046511103 (2^{42}-1) (type AMT$FILE_LENGTH).

Calls AMP$GET_FILE_ATTRIBUTES.

FILE_LIMIT

Meaning Maximum file length in bytes (preserved attribute).

Value Integer from 0 through AMC$FILE_BYTE_LIMIT, 4398046511103 (2^{42}-1) (type AMT$FILE_LIMIT).

NOTE

If the length of a keyed file reaches its file_limit value, the file is ruined (its structure loses its integrity). No file operations can be performed on a ruined file.

Default Value 4398046511103 (2^{42}-1).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

FILE_ORGANIZATION

Meaning File organization (preserved attribute).

Value One of the following keyed file organization identifiers (type AMT$FILE_ORGANIZATION):

AMC$INDEXED_SEQUENTIAL	Indexed-sequential organization.
AMC$DIRECT_ACCESS	Direct-access organization.

Default Value You must specify this attribute value when creating a keyed file because the default file organization is AMC$SEQUENTIAL.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

FILE_POSITION

Meaning Current file position (access information item).

Value One of these identifiers that apply to keyed files (type AMT$FILE_POSITION):

AMC$BOI	Beginning-of-information.
AMC$END_OF_KEY_LIST	End of the list of primary keys associated with the same alternate-key value.
AMC$EOR	End of record. (While an alternate key is selected, AMC$EOR indicates that the next record is associated with the same alternate-key value as the current record.)
AMC$EOI	End of information.

Calls AMP$FETCH_ACCESS_INFORMATION.

FORCED_WRITE

Meaning Indicates whether the system copies modified blocks to mass storage immediately after modification or allows modified blocks to remain in memory until the next flush or close request (preserved attribute).

Value One of the following identifiers (type AMT$FORCED_WRITE):

AMC$FORCED	The system writes each modified block to mass storage immediately after the block is modified.
AMC$FORCED_IF_STRUCTURE_CHANGE	The system writes modified blocks to mass storage immediately after any structure change to the file that affects more than one block.
AMC$UNFORCED	The system determines when to write modified blocks to mass storage. Modified blocks can remain in memory without a backup copy on mass storage.

Default Value AMC$FORCED_IF_STRUCTURE_CHANGE.

An AMP$FLUSH call copies the part of the file in memory to disk. AMP$FLUSH copies internal tables as well as data and index blocks. (A FORCED_WRITE copy does not copy internal tables.)

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

GLOBAL_ACCESS_MODE

Meaning Indicates the set of valid access modes for the file (returned attribute). (The access modes required for each keyed-file interface call are listed in table I-4-2.)

Value Set of any (including none) of the following constant identifiers (referenced using the set identifier PFTUSAGE_SELECTIONS [ ]):

PFC$READ	Read access.
PFC$SHORTEN	Shorten access.
PFC$APPEND	Append access.
PFC$MODIFY	Modify access.
PFC$EXECUTE	Execute access.

Default Value For an existing permanent file, the set of access modes is determined when the file is attached. For a temporary file or a new permanent file, the set includes all usage modes (read, modify, append, shorten, and execute).

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

GLOBAL_FILE_NAME

Meaning File name uniquely identifying the file (returned attribute). The system generates the name for the file when it creates the file. The global file name allows a program to determine whether files having different local file names are actually the same file.

Value Packed record (type OST$BINARY_UNIQUE_NAME).

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

GLOBAL_SHARE_MODE

Meaning Indicates the valid share modes for the file (returned attribute). For a permanent file, the share modes are specified when the file is attached. Temporary files cannot be shared. For more information, see Keyed-File Sharing in chapter I-2.

Value Set of any (or none) of the following constant identifiers. The attribute value is referenced using the set identifier PFTSHARE_SELECTIONS[ ].

PFC$READ	Read access.
PFC$SHORTEN	Shorten access.
PFC$APPEND	Append access.
PFC$MODIFY	Modify access.
PFC$EXECUTE	Execute access.

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

HASHING_PROCEDURE_NAME

Meaning Identification of the hashing procedure to be executed with the direct-access file (preserved attribute). (To read about hashing procedures, see chapter I-1.)

Value Pointer to a record identifying the hashing procedure to be executed with this file (^amt$hashing_procedure_name). The record has these fields:

NAME	Entry point name of the hashing procedure (pmt$program_name). All letters in the name must be specified as uppercase.
OBJECT_LIBRARY	File path to the object library containing the hashing procedure (amt$path_name, 256-character string). This feature is currently unimplemented; specify OSC$NULL_NAME as the field value.

Default Value The default hashing procedure provided by the system, AMP$SYSTEM_HASHING_PROCEDURE.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

INDEX_LEVELS

Meaning Target number of index levels (preserved attribute). The system uses the attribute value to calculate block size. The index_levels value is used only when an indexed-sequential file is created.

Value 1 through 15 (type AMT$INDEX_LEVELS).

Default Value 2.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

INDEX_PADDING

Meaning Percentage of space the system is to leave empty in each index block it creates during the first open of an indexed-sequential file (preserved attribute).

Value 0 through 99 (type AMT$INDEX_PADDING).

Default Value 0 (no padding).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

INITIAL_HOME_BLOCK_COUNT

Meaning Number of home blocks in the direct-access file (preserved attribute). (To read about direct-access file structure, see chapter I-1.)

Value Integer from 1 through amc$max_home_block_count ($2^{42}$-1) (type amt$initial_home_block_count).

Default Value None. You must specify a value for this attribute when defining a new direct-access file.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

KEY_LENGTH

Meaning Primary-key length in bytes (preserved attribute).

Value Integer (type AMT$KEY_LENGTH). (For files with embedded keys, the value cannot be greater than the minimum_record_length value.)

Default Value No default value. When opening a new keyed file, AMP$OPEN returns a fatal error if the attribute value is not defined.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

KEY_POSITION

Meaning Byte position in the record where the primary key begins (preserved attribute). This attribute is ignored for files with nonembedded keys.

The bytes in a record are numbered from the left, beginning with 0.

Value 0 through MAX_RECORD_LENGTH (type AMT$KEY_POSITION). The primary key must be within the record; thus, the sum of the key_position and key_length values cannot be greater than the max_record_length value.

Default Value 0 (beginning of the record).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

KEY_TYPE

Meaning Primary-key type (preserved attribute).

For direct-access files, the value specified for the key_type attribute is ignored. The primary-key type for a direct-access file is always uncollated.

Value One of the following identifiers (type AMT$KEY_TYPE):

AMC$UNCOLLATED_KEY
Order key values byte-by-byte according to the ASCII character set sequence (listed in appendix B). Key values can be positive integers or ASCII strings (1 through 255 bytes).

AMC$INTEGER_KEY
Order key values numerically. Key values are positive or negative integers (1 through 8 bytes).

AMC$COLLATED_KEY
Order key values according to a user-specified collation table (see the COLLATE_TABLE_NAME description in this table). Key values can be positive integers or ASCII strings (1 through 255 bytes).

Default Value AMC$UNCOLLATED_KEY.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

LAST_ACCESS_OPERATION

Meaning Indicates the last access request for this instance of open (access information item). (The code is set after the call checks that the file is open, but before it actually performs the operation.)

Value Value of type AMT$LAST_ACCESS_OPERATION. The following lists the file interface calls used with keyed files and the corresponding constant identifier declarations:

AMP$ABANDON_KEY_DEFINITIONS	amc$abandon_key_definitions
AMP$APPLY_KEY_DEFINITIONS	amc$apply_key_definitions
AMP$CLOSE	amc$close_req
AMP$CREATE_KEY_DEFINITION	amc$create_key_definition
AMP$CREATE_NESTED_FILE	amc$create_nested_file
AMP$DELETE_KEY	amc$delete_key_req
AMP$DELETE_KEY_DEFINITION	amc$delete_key_definition
AMP$DELETE_NESTED_FILE	amc$delete_nested_file
AMP$FETCH	amc$fetch_req
AMP$FLUSH	amc$flush_req
AMP$GET_KEY	amc$get_key_req
AMP$GET_LOCK_KEYED_RECORD	amc$get_lock_keyed_record
AMP$GET_LOCK_NEXT_KEYED_RECORD	amc$get_lock_next_keyed_record
AMP$GET_NESTED_FILE_DEFINITIONS	amc$get_nested_file_definitions
AMP$GET_NEXT	amc$get_next_req
AMP$GET_NEXT_KEY	amc$get_next_key_req

Value (Contd)	AMP$GET_NEXT_PRIMARY_KEY_LIST	amc$get_next_primary_key_list
	AMP$GET_PRIMARY_KEY_COUNT	amc$get_primary_key_count
	AMP$GET_SPACE_USED_FOR_KEY	amc$get_space_used_for_key
	AMP$LOCK_FILE	amc$lock_file
	AMP$LOCK_KEY	amc$lock_key
	AMP$OPEN	amc$open_req
	AMP$PUT_KEY	amc$put_key_req
	AMP$PUT_NEXT	amc$put_next_req
	AMP$PUTREP	amc$putrep_req
	AMP$REPLACE_KEY	amc$replace_key_req
	AMP$SELECT_KEY	amc$select_key
	AMP$SELECT_NESTED_FILE	amc$select_nested_file
	AMP$SKIP	amc$skip_req
	AMP$START	amc$start_req
	AMP$STORE	amc$store_req
	AMP$UNLOCK_FILE	amc$unlock_file
	AMP$UNLOCK_KEY	amc$unlock_key

Calls AMP$FETCH_ACCESS_INFORMATION.

LAST_OP_STATUS

Meaning Indicates whether the last access request is active or complete (access information item).

Value One of these identifiers (type AMT$LAST_OP_STATUS):

AMC$ACTIVE	Access request is active.
AMC$COMPLETE	Access request is complete.

Calls AMP$FETCH_ACCESS_INFORMATION.

LEVELS_OF_INDEXING

Meaning Number of index levels currently existing in the indexed-sequential file (access information item).

Value Integer from 0 through AMC$MAX_INDEX_LEVEL (type AMT$INDEX_LEVELS).

Calls AMP$FETCH_ACCESS_INFORMATION.

LOCK_EXPIRATION_TIME

Meaning Number of milliseconds between the time a lock is granted and the time that it could expire (preserved attribute).

Value Integer from 0 through 604,800,000 [1 week] (type amt$lock_expiration_time).

Default Value 0 (locks do not expire).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

MAX_BLOCK_LENGTH

Meaning Length in bytes of each keyed-file block (preserved attribute).

If specified, this value is used only when the keyed file is opened for the first time.

Value Integer from 1 through 16777215 (2^{24}-1). If the value is less than the maximum record length, the system increases it to that value. Then, if needed, it changes the value as follows:

- If the value is less than 2048, it is increased to 2048 (the minimum allocation unit).
- If the value is between 2048 and 65536, but it is not a power of 2, it is increased to the next power of 2 (4096, 8192, 16384, 32768, or 65536).
- If the value is greater than 65536, it is decreased to 65536.

Default Value For an indexed-sequential file, the system calculates an appropriate default value using the average_record_length, estimated_record_count, index_levels, and records_per_block values. For a direct-access file, it calculates the default value using the average_record_length and estimated_record_count values.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

MAX_RECORD_LENGTH

Meaning Maximum length of a file record in bytes (preserved attribute).

Value For keyed files, integer from 1 through 65497.

Default Value For keyed files, no default value is provided; AMP$OPEN returns a fatal error if the maximum record length has not been specified when the file is created.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

MESSAGE_CONTROL

Meaning Indicates the additional information to be written to the $ERRORS file (temporary attribute).

Value Set of any or none of the following identifiers. The attribute value is specified using the set identifier AMTMESSAGE_CONTROL[ ].

AMC$TRIVIAL_ERRORS	Nonfatal (trivial) errors logged (errors of severity ERROR).
AMC$MESSAGES	Informative messages logged.
AMC$STATISTICS	Statistics logged.

Default Value Null set (only fatal error messages are logged).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN, AMP$STORE.

MIN_RECORD_LENGTH

Meaning Minimum record length in bytes (preserved attribute).

Value For keyed files, integer from 0 though 65497, but not greater than the max_record_length value.

Default Value For ANSI fixed-length (F) records, the default value is the max_record_length value. For keyed files using embedded keys, the default value is the sum of the key_position and key_length values. Otherwise, the default value is 1.

NOTE

For variable-length records, it is recommended that you explicitly specify the minimum record length. The minimum record length must include:

- The primary-key field
- Any alternate-key fields (or corresponding sparse-key control characters)
- All alternate-key fields for an alternate key defined as a field in a repeating group which repeats a fixed number of times

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

NULL_ATTRIBUTE

Meaning Attribute identifier (AMC$NULL_ATTRIBUTE) that indicates that the content of the attribute record is to be ignored.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN, AMP$STORE.

NULL_ITEM

Meaning Access item identifier that indicates that the content of the attribute record is to be ignored.

Calls AMP$FETCH_ACCESS_INFORMATION.

NUMBER_OF_NESTED_FILES

Meaning Nested file count (access information item). Each keyed file has at least one nested file (named $MAIN_FILE).

Value Integer from 1 through amc$max_blocks_per_file.

Calls AMP$FETCH_ACCESS_INFORMATION.

OPEN_POSITION

Meaning Positioning required when the system opens the file (temporary attribute).

Value One of the following identifiers (type AMT$OPEN_POSITION):

AMC$OPEN_NO_POSITIONING or AMC$OPEN_AT_BOI	When the keyed file is opened, it is positioned to read the record with the lowest key value.
AMC$OPEN_AT_EOI	The file is positioned at its end-of-information. If the file is an old file and the only valid access mode to the file is append, the only valid open position is AMC$OPEN_AT_EOI.

Default Value For all files other than file OUTPUT, AMC$OPEN_AT_BOI. For file OUTPUT, AMC$OPEN_AT_EOI.

The open_position specified on a file reference overrides all specifications of that attribute except an open_position value specified by an AMP$OPEN call. For example, if a file is referenced as $USER.MY_FILE.$EOI, it is opened at its end-of-information unless the AMP$OPEN call specifies another open_position. For more information about file references, see the SCL Language Definition manual.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

PERMANENT_FILE

Meaning Indicates whether the file is permanent or temporary (returned attribute).

Value Boolean value.

TRUE	File is permanent.
FALSE	File is temporary.

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

PRIMARY_KEY

Meaning Pointer to a program variable in which the call is to return a primary-key value (access information item).

The primary-key value is for the record at which the preceding AMP$START call positioned the file or for the record read by the preceding AMPGET_NEXT_KEY, AMPGET_LOCK_NEXT_KEY, or AMP$GET_KEY call. This item can be returned only if the preceding call used an alternate key.

Value Cell pointer (type AMT$PRIMARY_KEY).

Calls AMP$FETCH_ACCESS_INFORMATION.

RECORD_LIMIT

Meaning Maximum number of records in the file (preserved attribute).

Value Integer from 1 through AMC$FILE_BYTE_LIMIT ($2^{42}$-1) (type AMT$RECORD_LIMIT).

Default Value AMC$FILE_BYTE_LIMIT ($2^{42}$-1).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

RECORD_TYPE

Meaning Record type for the file (preserved attribute).

Value One of the following identifiers (type AMT$RECORD_TYPE):

AMC$VARIABLE	CDC variable-length (V) records.
AMC$UNDEFINED	Undefined (U) records.
AMC$ANSI_FIXED	ANSI fixed-length (F) records.
	For keyed files, V and U records are processed the same (as variable-length records).

Default Value For keyed files, AMC$UNDEFINED.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

RECORDS_PER_BLOCK

Meaning Estimated number of records each data block should contain (preserved attribute).

The system uses the attribute value to calculate block size; it uses the value only when opening a new file. It does not use the value as a limit to the number of records that a block can contain.

Value Integer from 1 to AMC$MAX_RECORDS_PER_BLOCK (type AMT$RECORDS_PER_BLOCK).

Default Value 2.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

RESIDUAL_SKIP_COUNT

Meaning Number of units remaining to be skipped when the skip operation reached a file boundary (access information item). The residual skip count is the difference between the number of skip units requested and the number of units actually skipped.

Value Integer from 0 through AMC$FILE_BYTE_LIMIT (type AMT$RESIDUAL_SKIP_COUNT).

Calls AMP$FETCH_ACCESS_INFORMATION.

RETURN_OPTION

Meaning Indicates when the file is implicitly detached (returned) to the system (temporary attribute). (You can explicitly detach a file with a DETACH_FILE command or an AMP$RETURN call.)

Value One of the following identifiers (type AMT$RETURN_OPTION):

AMC$RETURN_AT_CLOSE	Detach when the task closes the file and the job does not have another instance of open for the file.

NOTE

The task closing the file does not receive notification that the file cannot be detached when the job has another instance of open of the file.

AMC$RETURN_AT_JOB_EXIT	Detach when the job terminates.

Default Value AMC$RETURN_AT_JOB_EXIT.

Calls AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

RING_ATTRIBUTES

Meaning Three ring numbers (r1, r2, and r3) defining the ring brackets of the file (preserved attribute).

- Write bracket: 1 through r1.
- Read bracket: 1 through r2.
- Execute bracket: r1 through r2.
- Call bracket: r2 + 1 through r3.

The ring numbers cannot be lower than the ring number of the caller that opens the file. If a new file is created by a file reference, its ring_attributes are those of the provider of the file reference specification.

Value Record with three integer fields R1, R2, and R3 (type AMT$RING_ATTRIBUTES).

Default Value All three ring numbers are the ring number of the AMP$OPEN caller. If the file has not yet been opened, the attribute value is undefined.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

SELECTED_KEY_NAME

Meaning Name of the currently selected key (access information item). If the primary key is the currently selected key, the name $PRIMARY_KEY is returned.

Value 31-character string, left-justified, blank-filled (type AMT$SELECTED_KEY_NAME). All letters in the name are returned in uppercase.

Calls AMP$FETCH_ACCESS_INFORMATION.

SELECTED_NESTED_FILE

Meaning Name of the currently selected nested file (access information item). By default, the currently selected nested file is $MAIN_FILE.

Value 31-character string, left-justified, blank-filled (type AMT$NESTED_FILE_NAME). All letters in the name are returned in uppercase.

Calls AMP$FETCH_ACCESS_INFORMATION.

Introduction to Sort/Merge II-1

What Sort/Merge Does II-1-2
  Data Flow II-1-2
Sort Keys II-1-3
  Multiple Keys II-1-3
Defining a Sort Key II-1-4
  Key Length and Position II-1-4
  Key Type II-1-5
  Collating Sequences II-1-6
  Numeric Data Formats II-1-7
  Sort Order II-1-12
Specifying the Record Length II-1-12
  Short Records II-1-13
  Invalid Records II-1-13
Example Program II-1-14

Introduction to Sort/Merge II-1

The CYBIL Sort/Merge interface is a set of CYBIL procedures. With these procedures, you can use NOS/VE Sort/Merge within your CYBIL program.

To include NOS/VE Sort/Merge within your CYBIL program, the program must include a sequence of procedure calls that specify the sort or merge request. The sequence of calls begins with either an SMP$BEGIN_SORT_SPECIFICATION call (for a sort request) or an SMP$BEGIN_MERGE_SPECIFICATION call (for a merge request). The sequence of calls ends with an SMP$END_SPECIFICATION call.

NOS/VE Sort/Merge use within your CYBIL program requires that the program include the Sort/Merge procedure and type declarations. The procedure and type declarations are stored in decks in the source library on file $SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE.

To copy the Sort/Merge procedure and type declarations into your program, you can copy one deck or several decks as follows:

- To copy a single deck containing all Sort/Merge procedure and type declarations, embed this SCU directive in your program:

```
*COPYC SMP$PROCEDURE_INTERFACE_PACKAGE
```

- To copy only those procedure and type declarations that are used in the program, embed an SCU *COPYC directive for each Sort/Merge procedure call used. The following are the directives required for a minimal sort specification:

```
*COPYC SMP$BEGIN_SORT_SPECIFICATION
*COPYC SMP$FROM_FILES
*COPYC SMP$TO_FILE
*COPYC SMP$KEY
*COPYC SMP$END_SORT_SPECIFICATION
```

To copy the procedure declarations from the system source library, store your source text (with the *COPYC directives embedded) as a deck in an SCU source library and expand it using an SCU EXPAND_DECK command. The EXPAND_DECK command specifies the system files containing the procedure and type declarations as alternate base libraries, as follows:

```
ALTERNATE_BASE=($SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE, ..
               $SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE)
```

This process is discussed in detail in the introduction to this manual.

NOTE

To execute a CYBIL program that uses Sort/Merge calls, you must add the following object library to the program library list:

```
$LOCAL.SMF$LIBRARY
```

What Sort/Merge Does

The purpose of sorting is to arrange items in order. The purpose of merging is to combine two or more sets of preordered items. Ordered information makes reports more meaningful and suggests critical relationships. Searches for information are faster with ordered lists.

The purpose of Sort/Merge is to arrange records in the sequence you specify. You describe the files of records that Sort/Merge is to sort and the order in which it is to sort them.

Sort/Merge:

- Sorts or merges records from as many as 100 files with one call to Sort/Merge.
- Sorts character and noncharacter key types.
- Can sort and merge variable-length (V) or fixed-length (F) records.
- Can read input records from and write output records to either sequential or indexed-sequential files. (The primary key of each indexed-sequential file must be embedded.)
- Can sort according to one of eleven predefined collating sequences, seven numeric formats, or a user-defined collating sequence.
- Can sum fields of records having equal keys.
- Can use owncode procedures to insert, substitute, modify, or delete records during Sort/Merge processing.

Data Flow

Sort/Merge reads input records from one or more local files or as supplied by an owncode routine. Records to be merged must be presorted. Records to be merged and summed must be pre-sorted and pre-summed.

Sort/Merge writes records to a single output file. The records can be processed by an owncode procedure.

Sort Keys

Sort or merge operations are based on the ordering of record fields in the data to be sorted or merged. These fields are called sort keys. This section discusses what sort keys are and how a key is defined.

A sort key is a field of data within each input record. Sort/Merge uses the contents of the sort key to determine the position of the record within the sorted sequence of records.

Data must be aligned correctly in a sort key field. Character data must be left-justified in the field, and numeric data must be right-justified in the field.

If sort keys extend beyond the length of the shortest record in the file, the sort is undefined. For example, if the records range from a minimum of 25 characters to a maximum of 80 characters, all sort keys must be in the first 25 characters for the sort to be defined.

Multiple Keys

A file can be sorted or merged on more than one sort key. The combined length of all key fields in a record cannot exceed 1023 bytes. The key fields cannot overlap.

The first key you specify is the most important key and is called the major sort key. This key is sorted or merged first. The keys you specify after the first key are of lesser importance and are called minor sort keys. The minor keys are numbered in the order they are specified.

For example, if three sort keys are specified, the first key is the major sort key (key number 1), the next key listed is a minor key (key number 2), and the third key is another minor key (key number 3).

When two or more records have an equal major key, Sort/Merge determines the order by looking at the subsequent minor keys in the following order: key number 2, key number 3, and so on. Sort/Merge compares the minor keys until either an unequal key is found, or until there are no more keys.

For example, university student records could be sorted using multiple sort keys. Assume each record includes the last name and first and middle initials, the student number, the date of birth, the field of study, the grade point average, and a code representing class (freshman, sophomore, junior, senior); all the fields are written with character data. The file could be maintained with the student number as the major key since records are normally retrieved by specifying the student number. The file can be sorted by the name in alphabetic order when a list of student names is needed.

When a university department needs to know which students are majoring in fields within the department, the file can be sorted on the field of study. The same sort can specify the name as a minor key so that records with the same field of study are also sorted in alphabetic order by the name. The file can be sorted by the class code as the major key and by the grade point average in descending numeric order as a minor key. This would produce a list of students sorted by class code with the students having the highest grade point average at the beginning of the list.

Defining a Sort Key

Each sort key to be used by the sort or merge request must be defined by a sort key definition on an SMP$KEY call. A sort key definition includes the following information:

Starting location of the key within the record

Key length

Type of data in the key field

Sort order

Key Length and Position

You define key field length and position by specifying the first byte of the field.

NOTE

When defining a Sort/Merge field, the leftmost byte in a record is counted as number 1.

For example, if you want to specify the name field of the university student record as a sort key, and the name field is the leftmost field in the record, you specify the first byte as 1. If the name field is 20 characters long, you specify the length as 20.

Sort/Merge interprets the integers you specify for key length and position as bit numbers when the key type (discussed later in this chapter) specifies bits; otherwise, byte numbers are assumed. The first bit is numbered 1; the key fields cannot overlap one another and cannot overlap sum fields.

Key Type

You specify the type of data in a key field with the name of a collating sequence or with the name of a numeric data format. The data in a key field can be character or noncharacter.

Character data is represented in the computer as ASCII code values. To indicate the key type for character data, you specify the name of a collating sequence.

Noncharacter data is represented in the computer as binary values, in packed decimal format, or in floating-point format. For numeric character data, you specify the name of a numeric data format.

The difference between the internal representation of character and noncharacter data is shown in figure II-1-1.

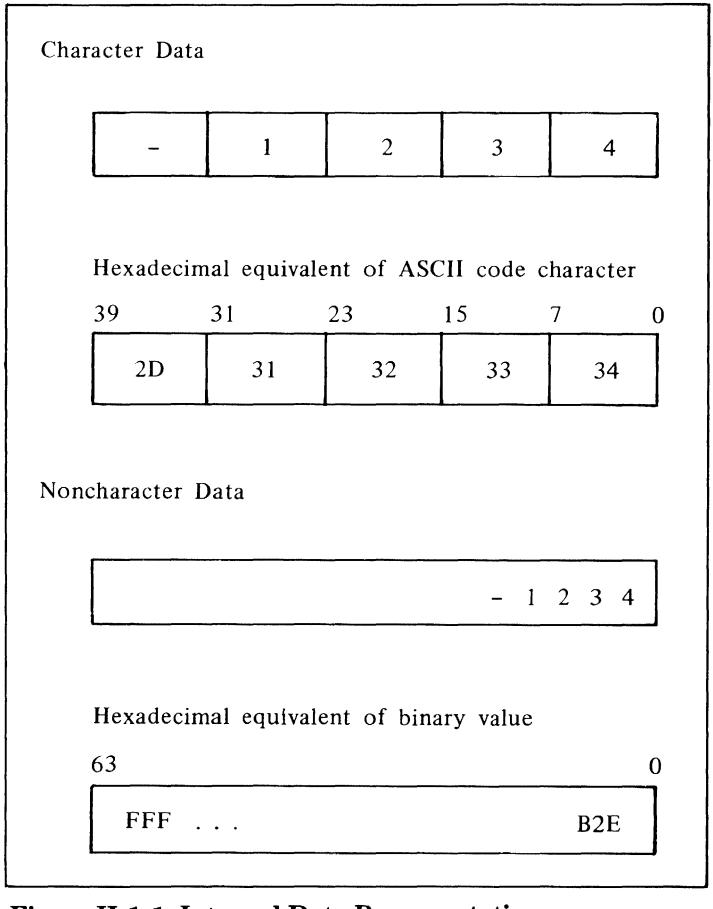


Figure II-1-1. Internal Data Representation

Table II-1-1 summarizes character and noncharacter data types and the associated sort key type.

Table II-1-1. Data in Sort Key Fields

Type	Internal Representation	Data in Field	Type Specified by	Data Ordered According to
Character	ASCII	Alphabetic	Name of a collating sequence	Specified collating sequence
		Numeric	Name of a numeric data format	Numeric value
Noncharacter	Binary value	Numeric	Name of a numeric data format	Numeric value
	Packed decimal numeric	Numeric	Name of a numeric data format	Numeric value

If a sort key field contains any characters that are not meaningful for the key type you specify (an alphabetic character in a field defined as a numeric key, for example), the sort order for that key field in that record is undefined. In the output file, the data for that key field in that record is also undefined. The record is still sorted according to other major sort keys you have specified, unless you have specified an exception file.

The collating sequences and numeric data formats you can specify are discussed in the following paragraphs.

Collating Sequences

A collating sequence determines the precedence given to each character in relation to the other characters. You specify the collating sequence that determines the sort order of character data. (Character data is represented as ASCII character codes.)

Sort/Merge defines six collating sequences: ASCII, ASCII6, COBOL6, DISPLAY, EBCDIC, and EBCDIC6. (NOS/VE defines five additional collating sequences, and you can define your own collating sequences.)

If you do not specify a collating sequence, ASCII is used. (Sort/Merge sorts fastest when using the ASCII collating sequence.)

The predefined collating sequences are listed in appendix D.

Numeric Data Formats

Numeric data can appear in a key field in one of the formats listed in table II-1-2. Numeric data can be signed or unsigned. For character numeric data that is signed, the sign can be a floating sign, an overpunch representation over the leading (leftmost) digit, a leading separate character, an overpunch representation over the trailing (rightmost) digit, or a trailing separate character.

Noncharacter numeric data can be signed or unsigned binary integers or normalized single precision floating-point numbers.

You define numeric key fields by specifying the first byte of the field and either the length of the field in bytes or the last byte of the field.

For BINARY_BITS and INTEGER_BITS data types, you specify the first bit position of the field and either the length of the field in bits or the last bit of the field.

For REAL data types, the key must be a full word aligned on a word boundary.

For other types except REAL, the fields start or stop on character boundaries.

Table II-1-2. Numeric Data Formats

Name	Data Type	Sign	Comments
BINARY	Binary integer	None	The field starts and ends on character boundaries. Data is ordered according to numeric value.
BINARY_BITS	Binary integer	None	The field does not start or end on character boundaries. Data is ordered according to numeric value.
INTEGER	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field starts and ends on character boundaries. Data is ordered according to numeric value.
INTEGER_BITS	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field does not start or end on character boundaries. Data is ordered according to numeric value.
NUMERIC_FS	Leading blanks, numeric characters	- sign for negative values; a + character is not allowed	The field contains leading blanks (leading zeros must be converted to blanks before calling Sort/Merge); if the value is negative, the rightmost leading blank must be converted to a minus sign. If the field contains no leading blanks or does not begin with a negative sign, the value must be positive. This format is equivalent to the FORTRAN I format, or the COBOL picture clause for zero suppressed editing of numeric item. Data is ordered according to numeric value.
NUMERIC_LO	Numeric characters	Leading overpunch	All characters are decimal digits except the leading character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_LS	Numeric characters	Leading separate	All characters are decimal digits except the leading character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_NS	Numeric characters	None	All characters are decimal digits. Data is ordered according to numeric value.
NUMERIC_TO	Numeric characters	Trailing overpunch	All characters are decimal digits except the trailing character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.

Continued

Table II-1-2. Numeric Data Formats *(Continued)*

Name	Data Type	Sign	Comments
NUMERIC_TS	Numeric characters	Trailing separate	All characters are decimal digits except the trailing character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.
PACKED	Packed decimal	Signed	Data is ordered according to numeric value.
PACKED_NS	Unsigned packed decimal	Unsigned	Data is ordered according to numeric value. PACKED_NS is the same as COBOL COMPUTATIONAL-3 with no sign.
REAL	Normalized binary real or single precision floating-point number of 64 bits	Signed	The field occupies a full computer word and is aligned on word boundaries. Data is ordered according to numeric value with all forms of zero ordered equally. The order of indefinite values is undefined. The order of infinite values is ordered as if its value were infinity (can be signed infinity). Double precision is not supported, but can be sorted by defining the upper part of the number as a primary real key and the lower part of the number as a secondary real key.

Signed Numeric Data

A floating sign is a negative sign embedded between leading blanks and the numeric characters. A floating sign can also be a negative sign followed by numeric characters. Leading zeros must be converted to blanks. Positive values in this format are not signed. The following examples are valid floating sign formats:

```
    - 1
      1
    - 0
      0
- 1 2 3
1 2 3 4
```

The following examples are invalid floating sign formats:

```
    0 1    Leading zero not allowed
  - 0 1    Leading zero not allowed
+ 1 2 3    Positive sign not allowed
           All-blank field not allowed
```

Diagnostic messages are not issued for invalid floating sign formats or invalid overpunches.

A negative sign overpunch is equivalent to overstriking a digit with a - , which is a punch in row 11. A positive sign overpunch is equivalent to overstriking a digit with a + , which is a punch in row 12.

When a signed overpunch digit is received as input, the digit is punched as indicated in the second column of table II-1-3. When a signed overpunch digit is entered from a terminal or displayed as output, the digit appears as indicated in the third column of table II-1-3. The hexadecimal value is in the fourth column.

Table II-1-3. Sign Overpunch Representation

Sign and Digit	Input Punch	Input/Output Representation	Hexadecimal Value
+0	0	0	30
+1	1	1	31
+2	2	2	32
+3	3	3	33
+4	4	4	34
+5	5	5	35
+6	6	6	36
+7	7	7	37
+8	8	8	38
+9	9	9	39
+0	12-0	{	7B
+1	12-1	A	41
+2	12-2	B	42
+3	12-3	C	43
+4	12-4	D	44
+5	12-5	E	45
+6	12-6	F	46
+7	12-7	G	47
+8	12-8	H	48
+9	12-9	I	49
-0	11-0	}	7D
-1	11-1	J	4A
-2	11-2	K	4B
-3	11-3	L	4C
-4	11-4	M	4D
-5	11-5	N	4E
-6	11-6	O	4F
-7	11-7	P	50
-8	11-8	Q	51
-9	11-9	R	52
+0	12-8-4	<	3C
+0	12	&	26
-0	12-8-7	!	21
-0	11	-	2D

Sort Order

Sort/Merge can sort a key in ascending or descending order. If you do not specify a sort order, Sort/Merge sorts the key in ascending order.

When sorting a numeric key in ascending order, Sort/Merge sorts the key values in order from lowest to highest. When sorting a numeric key in descending order, Sort/Merge sorts the key values in order from highest to lowest.

A character key is sorted according to the collating sequence you specify for the key. When sorting a character key in descending order, Sort/Merge sorts the key values in reverse order of the collating sequence you specify.

Specifying the Record Length

Sort/Merge accepts fixed-length (F) or variable-length (V) records. It can sort records up to 65,535 bytes long. The record type and record length are determined by the file attributes specified when the file is created.

The default maximum record length for both fixed-length (F) and variable-length (V) record types is 256 bytes. The default minimum record length for variable-length records is 0 bytes.

If the minimum record length for any Sort/Merge input file is 0, you must include an SMP$KEY call in the Sort/Merge call sequence. If you omit the SMP$KEY call and the minimum record length for any input file is 0, Sort/Merge attempts to use the 0 value (the smallest minimum record length of the input files) as the key length. But Sort/Merge cannot define a key of length 0, so it returns a fatal error.

Sort performance is best when the maximum record length is equal to the longest record to be sorted.

If the SORT or MERGE procedures do not specify any input or output files, Sort/Merge assumes that all records are provided by owncode procedures. In this case, you must specify the record length using either the SMP$OWNCODE_FIXED_RECORD_LENGTH or SMP$OWNCODE_MAX_RECORD_LENGTH procedure.

Short Records

Sort/Merge uses only those fields it needs to perform the sort or merge. For example, if the major key values are unequal, it does not use the minor key values. Similarly, if the key values are unequal, it does not use the sum values.

When Sort/Merge attempts to use a sum or key field beyond the end of the record, it sends an informative message and leaves the order of the short record undefined. If an exception file is specified for the sort or merge, it writes the short record to the exception records file and deletes it from the sort or merge.

Records could be too short because the system strips off all trailing blanks from variable-length (V) records. A record shortened by blank suppression cannot be sorted if its length is shorter than the minimum length required to read all key and sum fields. This is so even if you have specified a value for the maximum record length file attribute.

Blank suppression is demonstrated when a DISPLAY_FILE command displays empty records. An empty fixed-length (F) is shown as a record of length MAXRL filled with blanks. An empty variable-length (V) record is shown as having a length of zero, with no blanks present in the record. It is a zero-length record because all trailing blanks have been stripped from the record.

Zero-length records are not included in the sort or merge and are not counted in the number of records sorted or merged. The zero-length records are output as the last records, even if you specify the SMP$VERIFY, SMP$RETAIN_ORIGINAL_ORDER, SMPSUM, SMPEXCEPTION_FILE, or OWNCODE procedures. An informational message is issued stating how many zero-length records were read.

Invalid Records

Sort/Merge determines whether a key or sum field contains valid data when it attempts to use the data. Because Sort/Merge does not attempt to compare or sum the data in all fields, it does not validate all fields in a record; it only validates the data it uses.

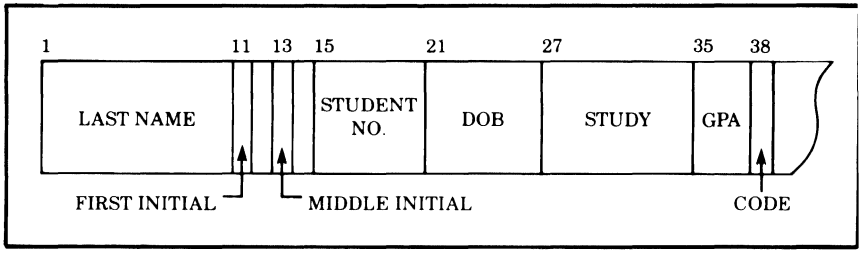
Sort/Merge copies the invalid records it finds to the exception records file (if one has been specified) and deletes the invalid records from the sort or merge.

Sort/Merge determines whether a key or sum field contains valid data when it attempts to use the data. If, when Sort/Merge attempts to compare or sum data from two records, it finds that one record contains invalid data, it then discards the invalid record and attempts to compare or sum the next record. It continues to do so until it finds a record containing valid data. Therefore, in the end cases, where either all records are invalid or the file contains only one record, one record will not be determined as invalid because it cannot be compared or summed with a valid record. So Sort/Merge always outputs at least one record, valid or invalid.

Example Program

The following example CYBIL program sorts a file on three keys.

The file is a file of student records. Each record has this format:



The records are first sorted on the field of study (byte positions 27 through 34 in each record), then on the class code (byte 38), and finally on the student's last name (bytes 1 though 10).

```
MODULE sort_files;

?? PUSH (LIST:= OFF) ??
*copyc smp$begin_sort_specification
*copyc smp$from_file
*copyc smp$to_file
*copyc smp$end_specification
*copyc pmp$exit
?? POP ??

VAR
  iarray: smt$info_array,
  file: string(19),
  status: ost$status;

PROGRAM sort;
    iarray[1]:=0;

{ Sequence of Sort/Merge calls }
    smp$begin_sort_specification (iarray, status);
    smp$from_file ('university_students', status);
    smp$to_file ('field_of_study', status);
    smp$key (27, 8, 'ascii', 'a', status);
    smp$key (38, 1, 'numeric_ns', 'a', status);
    smp$key (1, 10, 'ascii', 'a', status);
    smp$end_specification (status);

    IF status.normal <> true then
        pmp$exit (status);
    IFEND;

PROCEND sort;

MODEND sort_files;
```

Before a CYBIL program using Sort/Merge is compiled, the source text must be expanded to include the Sort/Merge procedure declarations. See the manual introduction for more information on this process.

Assuming that the source text is on file $USER.SOURCE_TEXT, the following command expand, compile, and execute the example program:

```
/create_source_library result=temporary_library
/source_code_utility base=temporary_library
sc/create_deck deck=sorting source=$user.source_text ..
sc../modification=original
sc/expand_deck deck=sorting ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit write_library=no
/cybil input=compile l=list b=lgo
/attach_file $user.university_students
/lgo
```

Assuming that these records are in file UNIVERSITY_STUDENTS, the program writes the records to the file FIELD_OF_STUDY in this order:

```
REYES      S L 100246031558ANTHRO  3341
MAYER      M I 100991122359ANTHRO  2882
CHARLES    S H 101418032459ANTHRO  2453
MARTIN     R C 100955082157Art     2891
NEECE      M L  99911121358Art     2291
NAKAMURA   S L 101529051260Art     2594
YEH        F L 102005120645Art     2764
BARTLETT   S S 100800100957Art     2735
COCHRAN    G L 100725111857BIO     3011
HOYO       J C 101925103060BIO     3014
.
.
.
KRUTZ      S T 100532010353POLISCI 1981
WALLIN     G E 101056041659POLISCI 3151
WARNES     D V 102116060861POLISCI 2814
WONG       S T 101001012755PSYCH   2152
LANGDON    M A 101754080549PSYCH   2013
LASEUR     P T 100678042256PSYCH   2233
SUGARMAN   B T 100528070457SOC     3501
SMITH      F R 101062120758SOC     2913
DOUGLAS    M L 101325071558UNDEC   2585
OKADA      N A 100103111750UNDEC   2225
```

Sort/Merge Procedure Calls II-2

Sort/Merge Procedure Call Use II-2-1
SMP$BEGIN_SORT_SPECIFICATION II-2-2
SMP$BEGIN_MERGE_SPECIFICATION II-2-4
SMP$FROM_FILE and SMP$FROM_FILES II-2-5
SMP$TO_FILE II-2-7
SMP$KEY II-2-9
SMP$DEFINE_USER_COLLATING_TABLE II-2-11
SMP$ERROR_FILE II-2-12
SMP$ERROR_LEVEL II-2-13
SMP$ESTIMATED_NUMBER_RECORDS II-2-15
SMP$EXCEPTION_RECORDS_FILE II-2-16
SMP$LIST_FILE II-2-17
SMP$LIST_OPTION II-2-18
SMP$LOAD_COLLATING_TABLE II-2-18.2
SMP$OWNCODE_FIXED_RECORD_LENGTH II-2-18.4
SMP$OWNCODE_MAX_RECORD_LENGTH II-2-19
SMP$OWNCODE_PROCEDURE_n II-2-20
SMP$RETAIN_ORIGINAL_ORDER II-2-21
SMP$COLLATING_x II-2-22
SMP$COLLATING_NAME II-2-22
SMP$COLLATING_CHARACTERS II-2-23
SMP$COLLATING_ALTER II-2-24
SMP$COLLATING_REMAINDER II-2-24
SMP$STATUS II-2-25
SMP$SUM II-2-26
SMP$VERIFY II-2-29
SMP$END_SPECIFICATION II-2-29

Sort/Merge Procedure Calls II-2

This chapter contains detailed descriptions of the Sort/Merge procedures in alphabetical order.

Sort/Merge Procedure Call Use

As described in chapter 1, a CYBIL program that calls Sort/Merge procedure must include the procedure declarations from decks in these files:

```
$SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE_SOURCE
$SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE
```

The Sort/Merge procedure and type declarations are listed in appendix C.

To execute a CYBIL program that uses Sort/Merge calls, you must add the following object library to the program library list:

```
$LOCAL.SMF$LIBRARY
```

The Sort/Merge procedures can be called in any order with two exceptions: SMP$BEGIN_SORT_SPECIFICATION or SMP$BEGIN_MERGE_SPECIFICATION must be the first procedure called, and SMP$END_SPECIFICATION must be the last procedure called. Sort/Merge collects processing information until SMP$END_SPECIFICATION is called; the sort or merge is then performed.

Unless stated otherwise, a procedure can be called only once during a sort or merge. Refer to chapter 3 for information on owncode procedures, which are mentioned in the descriptions of several of the Sort/Merge procedures.

Sort/Merge uses the maximum_record_length file attribute value in its processing. The maximum_record_length value is set when the file is created; the default maximum record length is 256 bytes.

With one exception, you can enter the Sort/Merge parameter values using uppercase, lowercase, or a combination of uppercase and lowercase letters. The one exception is owncode procedure names, which must be specified using all uppercase letters.

CYBIL owncode procedures that are loaded with the main program and referenced with the SMP$OWNCODE_PROCEDURE_n procedure call must be externally declared XDCL procedures.

SMP$BEGIN_SORT_SPECIFICATION

Purpose Signals the beginning of a sort calling sequence of procedure calls.

Format **SMP$BEGIN_SORT_SPECIFICATION (array, status);**

Parameters **array**: VAR of smt$info_array

Result array name; 1 to 31 letters, digits, or the special characters $ # @ _, beginning with a letter.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The SMP$BEGIN_SORT_SPECIFICATION procedure must be the first procedure called for a sort.
- The result array is a 0- through 16-element integer array in which Sort/Merge returns sort statistics and results to your program when the sort is completed. The result array is a single dimensional array.

  You set the first element of the result array to the number of elements (as many as 15) in the result array to receive information. If the first word is set to a value greater than 15 or less than 0, Sort/Merge issues a warning message and changes the value to 15 or 0, respectively.

  The type of result that is returned in each element of the result array is shown in table II-2-1.

Table II-2-1. Result Array Format

Array Element	Contents
1	Number of elements of results you want returned (0 through 15)
2	Number of records read from sort or merge input files
3	Number of records deleted by an owncode 1 routine
4	Number of records inserted by an owncode 1 routine
5	Number of records inserted by an owncode 2 routine
6	Number of records sorted or merged. (Does not include zero-length records or records written to the exception file.)
7	Number of records deleted by an owncode 3 routine
8	Number of records inserted by an owncode 3 routine
9	Number of records inserted by an owncode 4 routine
10	Number of records written to the exception file
11	Number of records deleted by an owncode 5 routine
12	Number of records combined by summing
13	Number of records written to the output file
14	Minimum record length. (Actual minimum record length of records from the file named by the SMP$FROM_FILE procedure and/or from owncode 1 and 2 routines.)
15	Average record length. (Total record length divided by the total number of records from the file named by the SMP$FROM_FILE procedure and/or from owncode 1 and 2 routines.)
16	Maximum record length. (Actual maximum record length of records from the file named by the SMP$FROM_FILE procedure and/or from owncode 1 and 2 routines.)

SMP$BEGIN_MERGE_SPECIFICATION

Purpose Signals the beginning of a merge calling sequence of procedure calls.

Format **SMP$BEGIN_MERGE_SPECIFICATION (array, status);**

Parameters

array: VAR of smt$info_array
Result array name; 1 to 31 letters, digits, or the special characters $ # @ _, beginning with a letter.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The SMP$MERGE_SORT_SPECIFICATION procedure must be the first procedure called for a merge.
- The result array is a 0- through 16-element integer array in which Sort/Merge returns merge statistics and results to your program when the merge is completed. The result array is a single dimensional array.

  You set the first element of the result array to the number of elements (as many as 15) in the result array to receive information. If the first word is set to a value greater than 15 or less than 0, Sort/Merge issues a warning message and changes the value to 15 or 0, respectively.

  The type of result that is returned in each element of the result array is shown in table II-2-1.

SMP$FROM_FILE and SMP$FROM_FILES

Purpose Specifies the input file or files from which the records to be sorted or merged are read.

Formats

SMP$FROM_FILES (file_name_array, status);

SMP$FROM_FILE (file_ref, status);

Parameters

file_ref: string(*)

Local file from which records are read for sorting or merging. The parameter must be a string or string variable specifying the name. Sort/Merge treats lowercase letters as being equal to uppercase letters.

file_name_array: array [*] of ost$name

Array of file names from which records are read for sorting or merging. Sort/Merge treats lowercase letters as being equal to uppercase letters.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- You can specify a maximum of 100 input (from) files using one or more procedure calls. The files are read in the order that you specify them. In addition, the files are not read past an embedded end-of-partition.
- SMP$FROM_FILE and SMP$FROM_FILES are separate procedures; each has its own procedure declaration deck. Call SMP$FROM_FILE to specify one input file name; call SMP$FROM_FILES to specify an array of input file names.
- When you are merging files, the records in each input file must be in sorted order. For a merge with summing, the records in each input file must be presummed as well as presorted.
- If you do not specify any SMP$FROM_FILE or SMP$FROM_FILES calls in the specification, records to be sorted or merged are read from the file OLD unless an owncode 1 procedure supplies records. However, file OLD is not assumed to exist and is not created by default if an owncode 1 procedure has been supplied.

Remarks (Contd)

- Specifying the file $NULL or an empty FROM file, both without an owncode 1 procedure specified, results in a null sort or merge. A null sort or merge has no records sorted or merged.

- Sort/Merge input files can have either sequential or indexed-sequential file organization and either variable-length (V) or fixed-length (F) record type.

  If an input file is an indexed-sequential file, its primary key must be embedded. If the primary key is nonembedded, Sort/Merge issues a fatal error and terminates.

SMP$TO_FILE

Purpose Specifies the file to which sorted or merged records are written if records are left after owncode procedure processing.

Format **SMP$TO_FILE (file_name, status);**

Parameters **file_name**: string(*)

Local file to which records are written. The parameter must be a string or string variable specifying the name. Sort/Merge treats lowercase letters as being equal to uppercase letters.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks:

- Sort/Merge closes the file when it completes the sort or merge.
- If a SMP$TO_FILE procedure is not called, records are written to the file NEW, an owncode 3 procedure can process all records, or records are written to file NEW, as changed by an owncode procedure. However, file NEW is not created by default if an owncode 3 procedure has been supplied. The file attributes are the NOS/VE defaults.
- The Sort/Merge output (SMP$TO_FILE) file can have either sequential or indexed-sequential file organization and either variable-length (V) or fixed-length (F) record type.
- If the output file is an indexed-sequential file, its primary key must be embedded. If the primary key is nonembedded, Sort/Merge issues a fatal error and terminates.
- Also, if the output file is an indexed-sequential file, the major sort key must be the primary key defined for the output file. The input records cannot have equal major sort key values because the primary-key values for the output file must be unique.

Remarks (Contd)

- If the output file is an indexed-sequential file, Sort/Merge checks the key_position, key_length, and key_type file attributes.
  - If the major sort key position does not match the key_position attribute value, Sort/Merge issues a fatal error and terminates.
  - If the major sort key length does not match the key_length attribute value, Sort/Merge issues a warning error and changes the major sort key length to match the primary-key length.
  - If the major sort key type does not match the key_type attribute value, Sort/Merge issues a warning error. It also changes the major sort key type if the key_type attribute specifies uncollated or integer keys. (It does not issue a warning or change the key type if the key_type attribute specifies collated keys.)
    - For uncollated keys, the major sort key type is changed to ASCII.
    - For integer keys, the major sort key type is changed to INTEGER.

  To read about indexed-sequential file attributes, see part I of this manual.

SMP$KEY

Purpose Specifies a single key field for the sort or merge.

Format **SMP$KEY (first, length, kind, ad, status);**

Parameters

first: integer

First byte or bit of the key field. Bytes or bits in a record are numbered from the left, beginning with 1.

length: integer

Number of bytes or bits in the field.

kind: string(*)

Kind of data in the key. For character data, the parameter specifies the name of a collating sequence; for numeric data, it specifies the name of a numeric data format. Sort/Merge treats uppercase letters as being equal to lowercase letters.

ad: char

Sort order; A or a for ascending, D or d for descending.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- You must specify all four parameters; there are no default values.
- The parameters first and length refer to bytes unless the key type is BINARY_BITS or INTEGER_BITS.
- You can call the SMP$KEY procedure as many as 106 times during a sort or merge to specify multiple sort keys.

  The significance of multiple keys corresponds to the order in which the keys are defined. Output records are sorted or merged according to the key field described by the first SMP$KEY procedure called, then according to the key field described by the second SMP$KEY procedure called, and so on.

  The total number of key characters must be no more than 1,023 eight-bit bytes. Key fields cannot overlap one another or a sum field and must be within the minimum record length.

Remarks (Contd)

- If the SMP$KEY procedure is not called, the following assumptions are made: the first byte is 1, the key length is the smallest minimum record length of any of the input files, the key type is the ASCII collating sequence, and the sort order is ascending.
- A warning error is issued if a key field contains invalid data. The warning error results in the following actions:
  1. The record is written to the exception records file if an exception records file was specified.
  2. The record is deleted from the sort or merge if an exception file was specified. If an exception records file was not specified, the record remains in the sort or merge, but its place in the sort order is undefined.
  3. A diagnostic message is issued, as controlled by the list options specification.
  4. The sort or merge continues normally.
- If the output (SMP$TO_FILE) file is an indexed-sequential file, the major sort key must be the embedded primary key defined for the output file. For details, see the SMP$TO_FILE procedure description.

SMP$DEFINE_USER_COLLATING_TABLE

Purpose Specifies a user-defined collation table.

Format

SMP$DEFINE_USER_COLLATING_TABLE (collating_sequence_name, weight_table, status);

Parameters

collating_sequence_name: string(*)

Name you choose to call the collating sequence produced by the collation table. This name is the name specified in a key field definition. Sort/Merge treats lowercase letters as being equal to uppercase letters.

weight_table: amt$collate_table

Array defining a collation table. The array has 256 elements; each element is an integer from 0 through 255 defining the collation weight of the corresponding ASCII character code.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- A sort or merge specification can include more than one SMP$DEFINE_USER_COLLATING_TABLE call.
- The following is an example of the declaration and initialization of a weight_table array. (It defines the predefined collating sequence OSV$DISPLAY64_FOLDED.)

```
VAR OSV$DISPLAY64_FOLDED: [#GATE,XDCL,READ]
AMT$COLLATE_TABLE:=
[ rep 33 of 45,
  54, 52, 48, 43, 51, 55, 56, 41, 42, 39, 37, 46,
  38, 47, 40, 27, 28, 29, 30, 31, 32, 33, 34, 35,
  36,  0, 63, 58, 44, 59, 57, 60,  1,  2,  3,  4,
   5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
  17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 49, 61,
  50, 62, 53, 60,  1,  2,  3,  4,  5,  6,  7,  8,
   9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
  21, 22, 23, 24, 25, 26, 49, 61, 50, 62,
  rep 129 of 45];
```

  For more information on collation tables, see appendix D.

- The collating sequence name specified on the call cannot be the name of a predefined collating sequence or another collating sequence you have already defined for the sort or merge.

SMP$ERROR_FILE

Purpose Specifies the file to which diagnostic messages are written.

Format **SMP$ERROR_FILE (file_name, status);**

Parameters **file_name**: string(*)

Local file name of the error file.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- Sort/Merge does not rewind the error file before or after it uses it.
- The file is written in V-type record format. If you specify the file $NULL with the SMP$ERROR_FILE procedure, diagnostic messages are not written.
- If you specify the same file for the listing file and for the error file, each error diagnostic message is written only once, not twice as it would be if the listing file and the error file were different and the messages were written to each file.
- In a batch job, both $LIST and $ERRORS are connected to OUTPUT. With $LIST and $ERRORS connected to the same file each error message is printed twice consecutively. To alleviate this situation you should always set one of the files to a nondefault value, using a value other than OUTPUT.
- If the SMP$ERROR_FILE procedure is not called, errors are written to file $ERRORS.

SMP$ERROR_LEVEL

Purpose Specifies the error level to be reported on the error file.

Format **SMP$ERROR_LEVEL ('limit', status);**

Parameters

limit: string(*)

Alphabetic character enclosed in apostrophes or a string variable containing the character (see table II-2-2).

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- An error can be one of the following levels:

Informational	An informational diagnostic results from a usage that is syntactically correct but questionable.
Warning	A warning diagnostic results when Sort/Merge finds an error but recovers by making assumptions about your attempt.
Fatal	A fatal diagnostic results when Sort/Merge cannot resolve an error. Sort/Merge treats error severity ERROR as a fatal error.
Catastrophic	A catastrophic error causes immediate Sort/Merge termination.

- The error levels that you can select are shown in table II-2-2. You can specify the alphabetic character enclosed in apostrophes in uppercase or lowercase letters. For example, if you specify W or w, any warning, fatal, and catastrophic error messages are reported.
- Errors are written to the file specified by the SMP$ERROR_FILE procedure. If the SMP$ERROR_LEVEL procedure is not called, all errors are reported, regardless of severity.

Table II-2-2. Error Level Specification Using the SMP$ERROR_ LEVEL Parameter

Error Level	Errors Reported
'I' or 'i'	Informational, warning, fatal, and catastrophic
'T' or 't'	(This is a nonstandard value and its use is not recommended)
'W' or 'w'	Warning, fatal, and catastrophic
'F' or 'f'	Fatal and catastrophic
'C' or 'c'	Catastrophic
'NONE' or 'none'	None

SMP$ESTIMATED_NUMBER_RECORDS

Purpose Provided for compatibility with NOS Sort/Merge 5; however, NOS/VE Sort/Merge does not use the specified value.

Format **SMP$ESTIMATED_NUMBER_RECORDS (value, status);**

Parameters

value: integer

Integer value indicating the estimated number of records to be sorted. The value can be from 1 through 16,777,215.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

SMP$EXCEPTION_RECORDS_FILE

Purpose Specifies the file to which invalid records are written.

Format **SMP$EXCEPTION_RECORDS_FILE (file_name, status);**

Parameters **file_name**: string(*)

Local file to which invalid records are written. The file name cannot be the same file name specified by the SMP$TO_FILE procedure. Sort/Merge converts the file name to all uppercase letters.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- If the SMP$EXCEPTION_RECORDS_FILE call specifies the $NULL file, Sort/Merge deletes all exception records. It does not write the exception records to an exception records file or to the output file.
- The records written to the exception records file include:
  - Records containing invalid key or sum field data
  - Records that caused an arithmetic overflow or underflow when their sum fields were summed.
  - Out-of-order merge input records if merge order checking was requested by an SMP$VERIFY call.
  - Records for which the system procedure AMP$PUT_NEXT returned an error when it attempted to write the record to the output (TO) file.
- The records in the exception file are deleted from the sort or merge. A summary of records written to the exception is printed in the error file named by the SMP$ERROR_FILE procedure call and in the list file.
- If you omit the SMP$EXCEPTION_RECORDS_FILE procedure call, Sort/Merge writes the invalid records to the output file. The invalid records are not written in a defined order.

SMP$LIST_FILE

Purpose Specifies the name of the list file.

Format **SMP$LIST_FILE (file_name, status);**

Parameters **file_name**: string(*)

Local file name of the listing file.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- Listing information includes the Sort/Merge version and level numbers, time and date, error messages, and statistics such as the number of records sorted or merged. If the SMP$LIST_FILE procedure is not called, the default list file is $LIST.
- If you specify the same file for the list file and for the error file, each error diagnostic message is written only once, not twice as it would be if the listing file and the error file were different and the messages were written to each file.
- In a batch job, both $LIST and $ERRORS are connected to OUTPUT. With $LIST and $ERRORS connected to the same file each error message is printed twice consecutively. To alleviate this situation you should always set one of the files to a nondefault value, using a value other than OUTPUT.

SMP$LIST_OPTION

Purpose Determines the type of information written to the listing file.

Format **SMP$LIST_OPTION (option, status);**

Parameters **option**: string(*)

Value indicating the listing information requested:

OFF	No additional information is to be written to the listing file.
NONE	Same as the OFF keyword.
S	Although it is a valid keyword, it has no meaning for this CYBIL procedure call. (It is meaningful on the SORT or MERGE command parameter.)
DE	Detailed exception information. A message is written for each occurrence that causes a record to be written to the exception records file. The DE keyword is valid only if you specify an exception records file; otherwise, an informational error message is issued. If you omit the DE keyword, messages are written only once per key, sum fields, or file that causes records to be written to the exception records file.
RS	Record statistics for the records sorted or merged. The statistics are from the result array; a message is written for each element of the array except for the first. Table II-2-1 lists the result array elements.
MS	Merge statistics for the records merged.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The minimum information Sort/Merge writes to the listing file is the page heading, error messages, the exception records file summary, and the number of records sorted or merged.
- You can specify only one list option with each SMP$LIST_OPTION procedure call, but the procedure can be called more than once.
- If you do not call the SMP$LIST_OPTION procedure, the list option used is S.

SMP$LOAD_COLLATING_TABLE

Purpose Loads a collation table, that is a weight table that defines a collating sequence. The table may be a NOS/VE predefined collation table or a user-defined collation table in an object library.

Format **SMP$LOAD_COLLATING_TABLE (collating_sequence_name, weight_table_name, status);**

Parameters

collating_sequence_name: string(*)

Name you choose to call the collating sequence produced by the collation table. This name is the name specified in a key field definition. Sort/Merge treats lowercase letters as being equal to uppercase letters.

weight_table_name: string(*)

Name of a predefined collation table or an object library entry point defining a collating sequence. Sort/Merge treats lowercase letters as being equal to uppercase letters.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- A sort or merge specification can include more than one SMP$LOAD_USER_COLLATING_TABLE call.
- The weight table must be loadable by PMP$LOAD.

  For more information on collation tables, see appendix D.
- Your collating sequence name cannot be the name of a predefined collating sequence or the name of a collating sequence you have already defined for the sort or merge.

Remarks (Contd)

- NOS/VE supplies 11 predefined collation tables. The following is a list of the predefined collation tables for these collating sequences:

Collating Sequences	Predefined Collation Table
CYBER 170 FTN5 default	OSV$ASCII6_FOLDED and OSV$ASCII6_STRICT
CYBER 170 COBOL5 default	OSV$COBOL6_FOLDED and OSV$COBOL6_STRICT
CYBER 170 63-character display code	OSV$DISPLAY63_FOLDED and OSV$DISPLAY63_STRICT
CYBER 170 64-character display code	OSV$DISPLAY64_FOLDED and OSV$DISPLAY64_STRICT
Full EBCDIC	OSV$EBCDIC
EBCDIC 6-bit subset	OSV$EBCDIC_FOLDED and OSV$EBCDIC_STRICT

- For more information on using and creating collation tables, see appendix D.

SMP$OWNCODE_FIXED_RECORD_LENGTH

Purpose Specifies the number of characters in fixed-length records entering the sort or merge from an owncode routine.

Format **SMP$OWNCODE_FIXED_RECORD_LENGTH (value, status);**

Parameters

value: integer

Fixed record length in bytes of all records supplied by any owncode procedure; maximum value is 65,535 bytes.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The integer you specify is the exact number of bytes in each record; a fatal error results if a record entering the sort from an owncode routine does not have the exact number of bytes.
- If the SMP$OWNCODE_FIXED_RECORD_LENGTH procedure is not called, records entering the sort from an owncode routine can be no longer than the longest allowed input or output record.
- If the sort has no input or output files (records to be sorted are supplied by an owncode routine and sorted records are processed by an owncode routine), you must specify one of the following procedures or else a fatal error results:

  SMP$OWNCODE_FIXED_RECORD_LENGTH
  SMP$OWNCODE_MAX_RECORD_LENGTH

- You cannot call both the SMP$OWNCODE_FIXED_RECORD_LENGTH procedure and the SMP$OWNCODE_MAX_RECORD_LENGTH procedure for the same sort.

SMP$OWNCODE_MAX_RECORD_LENGTH

Purpose Specifies the maximum length of any record entering the sort or merge from an owncode routine.

Format **SMP$OWNCODE_MAX_RECORD_LENGTH (value, status);**

Parameters

value: integer
Maximum record length in bytes of any record supplied by any owncode procedure; maximum value is 65,535 bytes.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The SMP$OWNCODE_FIXED_RECORD_LENGTH procedure is recommended if all records entering the sort from an owncode routine are the same length.
- The SMP$OWNCODE_MAX_RECORD_LENGTH procedure does not have to be called if the sort has an input or output file with a maximum record length at least as long as the record length specified by this procedure.
- If the SMP$OWNCODE_MAX_RECORD_LENGTH procedure is not called, records entering the sort from an owncode procedure can be no longer than the longest allowed input or output record.
- If the sort has no input or output files (records to be sorted are supplied by an owncode routine and sorted records are processed by an owncode routine), you must specify one of these procedures or else a fatal error results:

  SMP$OWNCODE_FIXED_RECORD_LENGTH
  SMP$OWNCODE_MAX_RECORD_LENGTH.

- You cannot call both the SMP$OWNCODE_FIXED_RECORD_LENGTH procedure and the SMP$OWNCODE_MAX_RECORD_LENGTH procedure for the same sort.

SMP$OWNCODE_PROCEDURE_n

Purpose Specifies an owncode routine to be executed each time a certain event occurs during the sort or merge.

Formats

SMP$OWNCODE_PROCEDURE_1
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_2
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_3
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_4
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_5
('procedure_name', status);

Parameters

procedure_name: string(*)
Owncode procedure name; 1 to 31 uppercase letters, digits, or special characters $ # @ _, beginning with a letter.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The procedure name is the name of the owncode routine. If you enter an owncode routine name in lowercase letters, Sort/Merge will not convert the name to uppercase letters. Use uppercase letters to name a routine.
- Sort/Merge loads the owncode procedures before it begins the sort or merge.
- If the SMP$OWNCODE_PROCEDURE_n procedure is not called, no owncode routine is executed.
- Owncode routines are described in detail in chapter 3.
- You cannot specify both the SMP$OWNCODE_PROCEDURE_5 and SMP$SUM procedure calls for the same sort or merge.
- You cannot specify an owncode 1 or 2 procedure for a merge.

SMP$RETAIN_ORIGINAL_ORDER

Purpose Specifies that records with equal sort keys are output in the same order as they are input.

Format **SMP$RETAIN_ORIGINAL_ORDER ('option', status);**

Parameters

option: string (*)
Indicates whether the input order is kept.

YES or Y	Keep the input order.
NO or N	Do not necessarily keep the input order.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- If the option is YES (or Y), and you specify more than one sort or merge input file with the SMP$FROM_FILE or SMP$FROM_FILES procedure, the order in which you specify the input files is the order in which records with equal keys are output.
- If the SMP$RETAIN_ORIGINAL_ORDER procedure is not called, records with equal keys can be output in either order.

SMP$COLLATING_x

Execution of the SMP$COLLATING_x procedures allow you to define your own collating sequence. A collating sequence specifies the sort or merge order for character data. You must define all 256 characters for the collating sequence or use the SMP$COLLATING_REMAINDER procedure. A collating sequence consists of a series of value steps from low value to high value. Each value step consists of at least one character representation. When a value step contains more than one character, all characters that are named within the step are collated equally.

A sequence of SMP$COLLATING_x procedures defines your collating sequence. Your collating sequence definition starts with the SMP$COLLATING_NAME procedure and ends by any procedure other than SMP$COLLATING_NAME, SMP$COLLATING_CHARACTERS, SMP$COLLATING_REMAINDER, or SMP$COLLATING_ALTER. You can define as many as 100 collating sequences by specifying a separate series of SMP$COLLATING_x procedures for each collating sequence.

SMP$COLLATING_NAME

Purpose Signals the start of your collating sequence definition and specifies the name of your collating sequence.

Format **SMP$COLLATING_NAME ('name', status);**

Parameters

name: string(*)
Your collating sequence name, 1 through 31 characters. The name must be a quoted literal specifying the sequence name.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- Your collating sequence name cannot be the same as the predefined collating sequence names and cannot be the same as a collating sequence you have already defined. Sort/Merge converts your sequence name to uppercase letters.

Remarks (Contd)

- Your collating sequence name is used as the key type in the SMP$KEY procedure call when records are sorted or merged according to your collating sequence. For example, the SMP$COLLATING_NAME procedure call shown below names a collating sequence.

```
SMP$COLLATING_NAME ('mysequence', status);
```

The following call defines a key field that uses the user-defined collating sequence MYSEQUENCE:

```
SMP$KEY (1, 10, 'mysequence', 'a', status);
```

SMP$COLLATING_CHARACTERS

Purpose Assigns collating positions to the characters in your collating sequence.

Format **SMP$COLLATING_CHARACTERS (char, status);**

Parameters

char: array [*] of char
One or more characters assigned to the collating position corresponding to the position of the call within the sequence of SMP$COLLATING_CHARACTERS (and SMP$COLLATING_REMAINDER) calls.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks The first SMP$COLLATING_CHARACTERS procedure call specifies the first value step or range of steps, the second SMP$COLLATING_CHARACTERS procedure call specifies the second value step or range of steps, and so on until your collating sequence is completely defined.

SMP$COLLATING_ALTER

Purpose Determines whether the characters in the value step defined by the preceding SMP$COLLATING_CHARACTERS call are altered in the output. If altered, all characters in the value step are output as the first character in the value step.

Format **SMP$COLLATING_ALTER ('option', status);**

Parameters **option**: string(*)

YES or Y Alter characters.

NO or N Do not alter characters.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

SMP$COLLATING_REMAINDER

Purpose Defines the position of the remainder value step in the collating sequence. The remainder value step consists of all characters that have not been included in value steps defined by SMP$COLLATING_CHARACTERS calls.

Format **SMP$COLLATING_REMAINDER ('option', status);**

Parameters **option**: string(*)

YES, Y, NO or N

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

SMP$STATUS

Purpose Specifies the name of the program variable in which Sort/Merge stores the most severe error that occurred during the sort or merge.

Format **SMP$STATUS (variable, status);**

Parameters

variable: VAR of integer
Name of the integer status for the Sort/Merge.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The error levels that are represented by the variable returned to the SMP$STATUS procedure are as follows:

Value	Error Level
0	No errors
10	Informational
20	Warning
30	Fatal
40	Catastrophic

  For example, if a 30 is returned to the SMP$STATUS procedure, a fatal error occurred during the sort or merge.

- If you call the SMP$STATUS procedure, Sort/Merge does not abort if a catastrophic error occurs before any data records are input. However, if Sort/Merge calls another product and an unrecoverable error results, an abnormal job termination does occur. Sort/Merge treats error severity ERROR as a fatal error.

SMP$SUM

Purpose Specifies one or more fields to be summed.

Format **SMP$SUM (first, length, 'stype', rep, status);**

Parameters

first: integer
First byte or bit of the sum field. (Bytes and bits are counted from the left, beginning with 1.)

length: integer
Number of bytes or bits in the sum field.

stype: string(*)
Name of a numeric data format.

rep: integer
Number of times the fields should be repeated to the right; a positive, nonzero integer.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The defined sum fields are summed when two records have equal keys. The records with equal keys are combined into one new record.

  The new record contains the equal keys and the summed fields. A data field that is not a key or sum field is written to the new record as a field from one of the old records.

- The location of a sum field is specified as the position as the first bit or byte in the field. Bits and bytes are numbered from the left in the record beginning with 1. The location is a byte position unless the numeric format of the sum field is BINARY_BITS or INTEGER_BITS.

- The maximum size of the BINARY, BINARY_BITS, INTEGER, INTEGER_BITS, PACKED, and PACKED_NS sum fields is one word. The maximum size of NUMERIC_LO, NUMERIC_LS, NUMERIC_TO, NUMERIC_TS, NUMERIC_NS, or NUMERIC_FS sum fields with a nonseparate sign is 17 digits. If the sum fields have a separate sign, the maximum size is 17 digits plus one digit for the sign.

Remarks (Contd)

- Sum fields can contain any type of numeric data, except REAL. Fields containing data in REAL format cannot be summed. See part II, chapter 1 for a list of the numeric data formats.
- The rep parameter specifies the number of consecutive sum fields defined by the SMP$SUM call. If the SMP$SUM call specified more the one field, the fields must be consecutive, must be the same length, and must contain the same type of numeric data.
- Sum fields cannot overlap one another.
- If a sum field contains no data because a record is short, the sum for that field is undefined.
- You can call SMP$SUM more than once for a Sort/Merge request. You can specify up to 100 sum fields per record.
- Sum fields and key fields cannot overlap. That is, the fields described as sum fields cannot also be key fields.
- You cannot specify both the SMP$RETAIN_ORIGINAL_ORDER and SMP$SUM procedures in the same sort or merge. If you specify both, a warning error occurs.
- You cannot specify both the SMP$SUM and SMP$OWNCODE_PROCEDURE_5 procedures in the same sort or merge. If you specify both, a warning error occurs.

Remarks (Contd)

- A fatal error is issued when a sum field contains invalid data or when an arithmetic overflow or underflow condition occurs as a result of summing two fields. An error due to invalid data leaves the contents of the sum fields undefined; an error due to an arithmetic overflow or underflow leaves valid data in the sum fields, but it may not be the original data.

  A fatal error results in the following actions:

  1. The record or records are written to the exception file if an exception file was specified. (If the error was due to invalid data in a sum field, one record is written; if the error was due to an arithmetic overflow or underflow, both records are written.)

  2. The record or records are deleted from the sort or merge if an exception file was specified. If an exception file was not specified, the record or records remains in the sort or merge, but their place in the sort order is undefined.

  3. A diagnostic message is issued depending on the list options specification.

  4. The sort or merge continues normally.

  If you do not include an SMP$SUM call in the sequence of Sort/Merge calls, records with equal key values are not combined into a single record.

SMP$VERIFY

Purpose Directs Sort/Merge to check that merge input records are in sorted order.

Format **SMP$VERIFY ('option', status);**

Parameters

option: string(*)
Indicates whether Sort/Merge is to verify the order of the merge input records.

YES or Y — Verify record order.

NO or N — Do not verify record order.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- If you omit the SMP$VERIFY call from a merge specification, Sort/Merge does not verify the record order.
- If you request merge order verification and Sort/Merge finds a merge input record out-of-order, Sort/Merge issues a diagnostic message. If an exception file was specified, Sort/Merge writes the out of order record to the exception file, deletes the record that is out of order, and continues merging.
- If you specify an SMP$VERIFY call for a sort, Sort/Merge issues a warning message but otherwise ignores the call.

SMP$END_SPECIFICATION

Purpose Terminates your sort or merge specification and initiates Sort/Merge processing.

Format **SMP$END_SPECIFICATION (status);**

Parameter

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks The SMP$END_SPECIFICATION procedure must be the last call in the Sort/Merge call sequence.

Owncode Procedures II-3

Specifying Owncode Procedures II-3-1

Owncode Procedure Parameters II-3-2

Owncode 1: Processing Input Records II-3-5

Owncode 2: Processing Input Files II-3-7

Owncode 3: Processing Output Records II-3-9

Owncode 4: Processing the Output File II-3-11

Owncode 5: Processing Records With Equal Keys II-3-12

Owncode Procedure Example II-3-13

Owncode Procedures II-3

You can write subprograms to insert, substitute, modify, or delete input and output records during Sort/Merge processing. Such a subprogram, called an owncode procedure, is executed each time the sort or merge reaches a certain point in Sort/Merge processing.

Sort/Merge passes a record to the owncode procedure, which processes the record. When the record is returned to Sort/Merge from the owncode procedure, Sort/Merge processes the record according to a code passed by the owncode procedure.

Owncode procedures can also supply the records to be sorted. When Sort/Merge is ready for a record, it calls the owncode procedure, which then passes a record to Sort/Merge.

Specifying Owncode Procedures

An SMP$OWNCODE_PROCEDURE_n call specifies the name of an owncode procedure Sort/Merge is to use; n is an integer from 1 through 5 that tells Sort/Merge at which point in processing the procedure is executed. The SMP$OWNCODE_PROCEDURE_n call is described in chapter 2.

Owncode procedures 1 and 2 can be called for a sort only; owncode procedures 3, 4, and 5 can be called for a sort or a merge.

SMP$OWNCODE_PROCEDURE_n calls are optional. Each SMP$OWNCODE_PROCEDURE_n call in the Sort/Merge sequence of calls must specify a different procedure name.

Use uppercase letters only when specifying a procedure name on an SMP$OWNCODE_PROCEDURE_n call. Sort/Merge does not convert lowercase letters in an owncode procedure name to uppercase letters.

You can write an owncode procedure using any NOS/VE programming language, including FORTRAN (subroutine subprograms), COBOL (subprograms compiled with COBOL SP=TRUE option), or CYBIL. The owncode procedure must be compiled and stored as a module in an object library.

Owncode procedures must either be loaded with the main program or be loadable from the program library list. To load an owncode procedure, Sort/Merge calls PMP$LOAD to load the procedure. PMP$LOAD then searches for the specified owncode procedure name in the directories of the object libraries in the program library list.

CYBIL owncode procedures that are loaded with the main program and referenced with SMP$OWNCODE_PROCEDURE_n procedure calls must be declared XDCL procedures.

For Sort/Merge to use an object library containing one or more owncode procedures, the object library file must be in the program library list. To add a file to the program library list before executing the CYBIL program, execute a SET_PROGRAM_ATTRIBUTES command.

For detailed information on creating object libraries, see the SCL Object Code Management Usage manual. The example at the end of this chapter stores an owncode procedure in an object library.

Owncode Procedure Parameters

Sort/Merge communicates with an owncode procedure via parameters. The parameters are passed each time Sort/Merge executes the owncode procedure.

Table II-3-1 summarizes the owncode procedures and the parameters passed. Some parameters cannot be omitted; see table II-3-1 for the required parameters.

The parameters passed between Sort/Merge and your owncode procedures are:

VAR return_code: integer
Code altered by an owncode procedure and returned to Sort/Merge

VAR reca: string (*)
Contents of a record

VAR rla: integer
Record length of a record

VAR recb: string(*)
Contents of a second record (owncode 5 procedure only)

VAR rlb: integer
Record length of a second record (owncode 5 procedure only)

Table II-3-1. Owncode Procedure Summary

Procedure Type	Processing	Parameters Passed: Return_ Code	reca	rla	recb	rlb	Return_code Value
Owncode 1	Input records	x	x	x			0 Sort current record.
							1 Delete current record.
							2 Insert new record.
							3 Terminate input from current file.
Owncode 2	Input files	x	x	x			0 Begin processing next input file, if any.
							1 Insert new record.
Owncode 3	Output records	x	x	x			0 Output current record.
							1 Delete current record.
							2 Insert new record.
							3 Terminate output.
Owncode 4	Output file	x	x	x			0 Sort or merge is complete.
							1 Insert new record.
Owncode 5	Equal keys	x	x	x	x	x	0 Retain both records.
							1 Replace both records with new record.

x = required parameter

The return_code parameter is passed by Sort/Merge to an owncode procedure as an integer with value 0. The return_code parameter can be altered by the owncode procedure to the integer value 1, 2, or 3, or the parameter can be left unchanged. The value returned to Sort/Merge by this parameter indicates a specific action to be taken by Sort/Merge. A return_code value that is not defined causes a fatal error. The meanings of the various return_codes are discussed later in this chapter.

The reca parameter is a variable used to pass the current record; except for the current record, the contents of reca are undefined. The rla parameter passes an integer value indicating the number of characters in current record passed by reca.

The recb and rlb parameters are used only by an owncode 5 procedure; an owncode 5 procedure processes two records with equal keys. The first record is in reca, with length rla characters. The recb variable passes the second record with length rlb characters.

The allowed length of records passed to and from an owncode procedure depends on how you have specified the record length, as follows:

- If you have specified the SMP$OWNCODE_FIXED_LENGTH procedure, the number of bytes in the current record must equal the SMP$OWNCODE_FIXED_LENGTH value.
- Otherwise, the maximum record length is determined as the largest value of the following:
  - The maximum_record_length file attribute values of the input or output files
  - The record length value specified by an SMP$OWNCODE_MAX_RECORD_LENGTH procedure call.

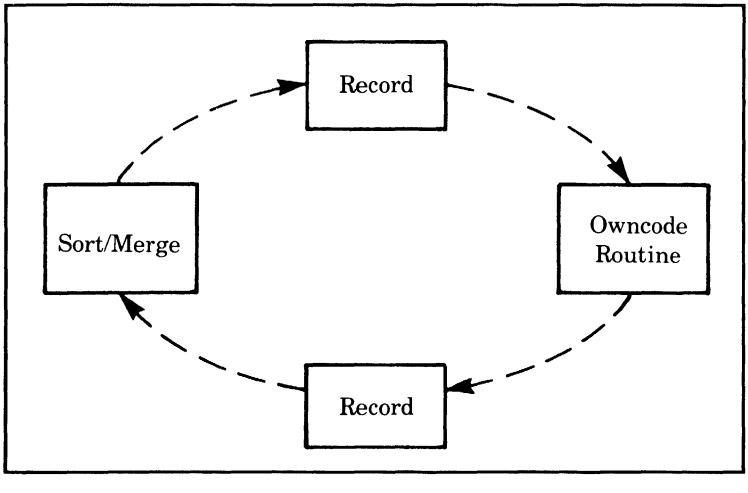
  In this case, the number of bytes in each record can range from 1 through the maximum record length value.

Either the owncode maximum record length or owncode fixed length must be specified if there are no input or output files.

An rla or rlb parameter value that does not correspond to a record specification causes an error.

The contents of the reca, rla, recb, and rlb variables can be altered by an owncode procedure; the routine can pass a different record back to Sort/Merge in reca or recb, and the number of characters in the record can be different.

The record movement from Sort/Merge to an owncode procedure and back to Sort/Merge is shown below.



Owncode 1: Processing Input Records

An SMP$OWNCODE_PROCEDURE_1 procedure call specifies an owncode 1 procedure. Sort/Merge executes an owncode_procedure_1 call each time it reads an input record.

An owncode 1 procedure is used only with a sort request; specification of an owncode 1 procedure in a merge request returns a fatal error.

One or More Input Files Specified

If you specify one or more input files or the value $NULL on the SMP$FROM_FILES or SMP$FROM_FILE procedure call, the owncode 1 procedure is executed after reading each record. The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, reca contains the record, and rla is the record length in characters.

After owncode processing of the record, control returns to Sort/Merge, which processes the record passed back in reca according to the return_code value set by the owncode 1 procedure. The record passed back to Sort/Merge in reca can be different from the record originally passed to the procedure.

The return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The record passed back to Sort/Merge in reca is sorted. The owncode 1 procedure is executed again with reca as an empty array and with rla=0.

1 The record passed back to Sort/Merge in reca is deleted.

2 An additional record is inserted into the sort. The record in reca is entered into the sort, and the owncode 1 procedure is executed again with reca and rla set to the record that just entered the sort.

3 Input from the current file is terminated. The record in reca and any remaining records in the file are not sorted. If more input files are specified, records are read from the next input file. The owncode 1 procedure is executed for each record read from the next file.

Input Files Not Specified

If you do not specify any input files (you omit the the SMP$FROM_FILES call from the call sequence), the owncode 1 procedure is executed when Sort/Merge is ready for another record to process. The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, reca is an empty array with enough space for the largest record, and rla is 0.

When control is returned to Sort/Merge from the owncode 1 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The record passed back to Sort/Merge in reca is sorted. The owncode 1 procedure is executed again with reca as an empty array and with rla=0.

2 An additional record is inserted into the sort. The record in reca is entered into the sort, and the owncode 1 procedure is executed again with reca and rla set to the record that just entered the sort.

3 Input is terminated; anything in reca or rla is ignored.

Owncode 2: Processing Input Files

An SMP$OWNCODE_PROCEDURE_2 procedure call specifies an owncode 2 procedure. Sort/Merge executes the owncode 2 procedure to process input files.

An owncode 2 procedure is used only with a sort request; specification of an owncode 2 procedure in a merge request returns a fatal error.

One or More Input Files Specified

If you specify one or more input files or the value $NULL with the SMP$FROM_FILES or SMP$FROM_FILE procedure call, the owncode 2 procedure is executed after input from a file has terminated. Input terminates when end-of-partition is found, when end-of-information is found, or when an owncode 1 procedure passes a return_code value of 3 to Sort/Merge.

The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, and reca and rla are passed as a null record; reca is an empty array, and record length is 0.

When control is returned to Sort/Merge from the owncode 2 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 Processing of the next input file, if any, begins.

1 An additional record is inserted into the sort after the last record. The record inserted is the first rla characters in reca, which have been provided by the procedure. The owncode 2 procedure is executed again.

Input Files Not Specified

If you do not specify any input files (you omit the the SMP$FROM_FILE call from the call sequence), the owncode 2 procedure is executed after an owncode 1 procedure has terminated input.

The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, and reca and rla are passed as a null record.

When control is returned to Sort/Merge from the owncode 2 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 Signals the end of input.

1 An additional record is inserted into the sort after the last record. The record inserted is the first rla characters in reca, which have been provided by the procedure. The owncode 2 procedure is executed again.

Owncode 3: Processing Output Records

An SMP$OWNCODE_PROCEDURE_3 procedure call specifies an owncode 3 procedure. Sort/Merge executes the owncode 3 procedure to process output records.

Output File Specified

If you specify an output file or the value $NULL with the SMP$TO_FILE procedure call, the owncode 3 procedure is executed each time a record is ready to be written to the output file.

The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, reca is the output record, and rla is the record length in characters.

After owncode processing of the record, control returns to Sort/Merge, which processes the record passed back in reca according to the return_code value set by the owncode 3 procedure. The return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The record passed back to Sort/Merge in reca is written to the output file.

1 The record passed back to Sort/Merge in reca is not written to the output file.

2 An additional record is written to the output file. The record in reca is written out, and the owncode 3 procedure is executed again with reca and rla set to the original record.

3 Output to the file is terminated. The record in reca is not written out. If an owncode 4 procedure is specified, the procedure is executed; otherwise, the sort or merge is terminated.

Output File Not Specified

If you do not specify an output file (you omit the the SMP$TO_FILES call from the call sequence), the owncode 3 procedure performs all output processing. Each output record is passed to the owncode 3 procedure. Sort/Merge does not write the record to an output file.

The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, reca is the record, and rla is the record length in characters.

When control is returned to Sort/Merge from the owncode 3 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

1 Owncode 3 is called again.

3 Output is terminated. If an owncode 4 procedure is specified, the procedure is executed; otherwise, the sort or merge is terminated.

Owncode 4: Processing the Output File

The SMP$OWNCODE_PROCEDURE_4 procedure specifies an owncode 4 procedure. Sort/Merge executes the owncode 4 procedure to process the output file.

Output File Specified

If you specify an output file or the value $NULL with the SMP$TO_FILE procedure call, the owncode 4 procedure is executed after the last record has been written to the output file.

The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, and reca and rla are passed as a null record.

When control is returned to Sort/Merge from the owncode 4 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The sort or merge is completed.

1 An additional record is inserted after the last record. The record inserted is the first rla characters in reca. The owncode 4 procedure is executed again with return_code, reca, and rla.

Output File Not Specified

If you do not specify an output file (you omit the the SMP$TO_FILES call from the call sequence), the owncode 4 procedure is executed after an owncode 3 procedure has terminated output.

The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, and reca and rla are passed as a null record.

When control is returned to Sort/Merge from the owncode 4 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The sort or merge is completed.

1 This value is meaningless because no output file has been specified.

Owncode 5: Processing Records With Equal Keys

An SMP$OWNCODE_PROCEDURE_5 procedure call specifies an owncode 5 procedure. Sort/Merge executes the owncode 5 procedure when it encounters two records with equal key values during a sort or merge.

The SMP$OWNCODE_PROCEDURE_5 procedure can be called at any time during the sort or merge whenever Sort/Merge detects duplicate records.

The return_code, reca, rla, recb, and rlb parameters are passed to the procedure by Sort/Merge. The return_code is 0; reca and recb contain the first and second records, respectively, and rla and rlb contain the record lengths in characters of the first and second records, respectively.

After the owncode 5 procedure processes the two records, control is returned to Sort/Merge. Sort/Merge then processes the records according to the return_code value set by the owncode 5 procedure. The return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The first rla characters of reca are accepted as the first record; the first rlb characters of recb are accepted as the second record (the records and record lengths passed back to Sort/Merge can be different from the records and record lengths passed to the owncode procedure).

1 One duplicate record is deleted. The other record is replaced with the first rla characters of reca.

If you call the SMP$RETAIN_ORIGINAL_ORDER procedure in a sort with an owncode 5 procedure, the record that first entered the sort is passed to the owncode 5 procedure as reca; otherwise, either of the two records with equal keys could be passed to the procedure as reca.

The owncode 5 procedure can control the order in which the two records are written to the output file. The record returned to Sort/Merge as reca is written to the output file before the record is returned as recb.

Owncode Procedure Example

An owncode 3 procedure written in FORTRAN is shown below. The procedure deletes the first record in a file. The variable COUNT keeps track of the number of times the owncode routine is entered.

```
SUBROUTINE OWNCODE (retcode,reca,rla)
INTEGER retcode, rla, count
CHARACTER reca*38
DATA count /0/

count = count +1

IF (count.eq.1) THEN
   retcode = 1
ELSE
   retcode = 0
ENDIF

RETURN
END
```

For detailed information on placing a subroutine into an object library, see the SCL Object Code Management Usage manual. The commands to place the example subroutine OWNCODE into the object library file $USER.OWN_LIBRARY are as follows:

```
/fortran input=owncode
/create_object_library
COL/add_module library=$local.lgo
COL/generate_library library=$user.own_library
COL/quit
```

The following command displays the contents of $USER.OWN_LIBRARY:

```
/display_object_library library=$user.own_library ..
../display_option=entry_point

Display of object library - OWN_LIBRARY

OWNCODE                          - load module

entry points
~~~~~~~~~~~~


OWNCODE
```

The following command adds $USER.OWN_LIBRARY to the program library list:

```
/set_program_attribute add_library=$user.own_library
```

After executing these commands, a CYBIL program can be executed in which the subroutine OWNCODE can be called from a sequence of Sort/Merge procedure calls such as:

```
smp$begin_sort_specification (iarray, status);
smp$from_file ('university_students', status);
smp$to_file ('field_of_study', status);
smp$key (1, 10, 'ascii', 'a', status);
smp$owncode_procedure_3 ('owncode', status);
smp$end_specification (status);
```

D

Data Block

A keyed-file block in which data records are stored. Contrast with Index Block.

Data-Block Split

The process of creating two or three data blocks from an existing data block when a record to be written does not fit into the remaining space of the existing block.

Deck

A sequence of lines in a source library that can be manipulated as a unit by the Source Code Utility (SCU).

Default Value

The value used for the parameter value if no value is explicitly specified.

Descending Sort Order

Used with the Sort/Merge interface to mean sorting key values from highest to lowest value. Contrast with Ascending Sort Order.

Direct-Access File Organization

A keyed-file organization in which records are accessed directly by hashing the primary-key value. Records can be accessed sequentially, but the records are not returned in sorted order. Contrast with Indexed-Sequential File Organization.

Duplicate Key Value

The situation detected when a record to be written to the file has a key value that matches a key value already in the file (or another value for the alternate key in the same record). It can also be detected during application of a new alternate-key definition to a file.

Duplicate Key Value Control

The alternate-key attribute that indicates whether duplicate values are allowed for the key and, if so, how the duplicates are ordered.

E

EBCDIC

The abbreviation for extended binary-coded decimal interchange code, an 8-bit code representing a coded character set.

Embedded Key

Key that is part of the data in each record. (Alternate keys are always embedded.) Contrast with Nonembedded Key.

End-Of-Information (EOI)

The point at which data in a file ends. For a keyed file, the EOI file position means that the file is positioned after the record with the highest key value.

Entry Point

A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

Exception Records File

As used with the Sort/Merge interface, a file to which invalid records are written before the records are removed from the sort or merge.

External Reference

A reference in one program unit to an entry point in another program unit.

F

F Record Type

Fixed-length records, as defined by the ANSI standard.

Field

A subdivision of a record.

File

A collection of information referenced by a name.

File Attribute

A characteristic of a file. Each file has a set of attributes that define the file structure and processing limitations.

File Cycle

A version of a file. All cycles of a file share the same file entry in a catalog. The file cycle is specified in a file reference by its number or by a special indicator, such as $NEXT.

File Organization

The file attribute that determines the record access method for the file. See Sequential File Organization, Byte-Addressable File Organization, and Keyed File Organization.

File Position

The location in the file at which a subsequent sequential read or write operation would begin.

File Reference

An SCL element that identifies a file and optionally the file position to be established prior to use.

Flush Request

A program request to write to the file device the parts of a file that have been modified in memory since the last time the file was written. For keyed files, the file device is always disk; for sequential files, the flush request can write to disk or to an interactive terminal.

Flushing

The process of writing to disk any parts of a file whose images in real memory have been altered or expanded, if the alteration or expansion has not yet been made on disk. Flushing does not alter the logical status or position of a file.

H

Hashing Procedure

The procedure used to relate a primary-key value to a home block number in a direct-access file. The procedure is executed for each file request that specifies a key value.

Home Block

A unit of space in a direct-access file that can be accessed directly. If possible, data records are stored in home blocks. Contrast with Overflow Blocks.

I

Index Block

An indexed-sequential file block in which index records are stored. Contrast with Data Block.

Index-Block Split

The process of creating two index blocks from an existing index block when a record to be written does not fit into the remaining space of the existing block.

Index Level

A rank in the index-block hierarchy in an indexed-sequential file. For the pointer to a data record to be found, an index block must be searched at each index level.

Index Level Overflow

The condition when a record cannot be written to a file because writing the record would require addition of another index level and the file already has 15 index levels.

Index Record

A record in an index block that associates a key value with a pointer to either a data block or an index block in the next-lower level of the index hierarchy.

Indexed-Sequential File Organization

A keyed-file organization in which records can be read sequentially, ordered by key value, or read randomly by a key value.

Instance of Open

A particular opening of a file as distinguished from all other openings of the file. The system assigns each instance of open a unique file identifier. Closing the file ends the instance of open.

Integer Key

The key type that orders key values numerically. The key values can be positive or negative integers. Contrast with Collated Key and Uncollated Key.

J

Job

A set of tasks executed for a user name.

K

Key

For Sort/Merge, a key is a part of a record used to determine the position of the record within a sorted sequence of records.

In a keyed file, a key is a value associated with a record as a means of accessing records. It may be a field in the record. See Primary Key and Alternate Key.

Key List

The sequence of primary-key values associated with an alternate-key value in an alternate index. If the alternate key does not allow duplicate values, each key list contains only one value. Otherwise, each key list contains a primary-key value for each record that contains the alternate-key value.

Key Type

The kind of data in a key.

For Sort/Merge, a key type is the name of a numeric data format or collating sequence.

For a keyed file, the possible key types are uncollated, collated, and integer.

Keyed-File Organization

A file organization that provides for record access by a key value. Currently, the keyed-file organizations are indexed-sequential and direct-access.

L

Library

See Source Library and Object library.

Local File

A file that is accessed via the $LOCAL catalog. See also File, Path, and Local Path.

Local File Name

The name used by an executing job to reference a file while the file is assigned to the job's $LOCAL catalog. Only one file can be associated with a given name in a job; however, in one job, a file can have more than one instance of open by that name.

Local Path

Identifies a local file as follows:

$LOCAL.file_name

Lock

The method by which an instance of open makes a primary-key value (or, for a file lock, all primary-key values) inaccessible to other instances of open of the file.

M

Major Key

The leftmost part of a key. The number of bytes to be used is specified as the major key length. A major key can be used to position or read a keyed file.

Major Sort Key

As used with the Sort/Merge interface, a sort key that is the most important and is specified first.

Mass Storage

Disk storage.

Merge

The process of combining two or more presorted files.

Minor Sort Key

As used with the Sort/Merge interface, a sort key that is specified after the major sort key on a SORT or MERGE command or in a procedure call. Minor keys are sorted after the major sort key.

Module

A unit of code. An object module is the unit of object code corresponding to a compilation unit. A load module is a unit of object code stored in an object library.

When using the Debug utility, module refers to a program unit.

N

Nested File

File defined within a keyed file. A nested file is recognized and used by the keyed-file interface; it is not recognized or used by the NOS/VE file system.

Nonembedded Key

A primary key that is not part of the record data. Contrast with Embedded Key.

Null Suppression

Alternate-key attribute indicating that records with null alternate-key values are not included in the alternate index.

O

Object Code

Executable code produced by the compiler.

Object Library

A library of modules that the system can load and execute as needed.

Open Operation

A set of preparatory operations performed on a file before input and output can take place.

Open Request

A program request notifying the system that the program wants to access file data.

Overflow Block

Unit of space in a direct-access file to which a record is written when its home block is full. See also Home Block.

Owncode Procedure

A user-written module, executed by Sort/Merge, that inserts, substitutes, modifies, or deletes records.

P

Padding

Space deliberately left unused in each block created during the initial open of a keyed file. Keyed-file blocks are padded to allow easy insertion of records after creation of the file.

Path

Identifies a file. A path may include the family name, user name, subcatalog name or names, and file name.

Permanent File

A file preserved by NOS/VE across job executions and system deadstarts. A permanent file has an entry in a permanent catalog. See File.

Piece

One of the fields of a concatenated alternate key.

Primary Key

The required key in a keyed file. Primary-key value must be unique in the file. See also Alternate Key.

Program-Library List

The list of object libraries searched for modules during program loading. A program-library list search is required to load a collation table module or a Sort/Merge owncode procedure module.

R

Random Access

The process of reading or writing a record in a file without having to read or write the preceding records; applies only to mass storage files. Contrast with Sequential Access.

Record

A unit of data that can be read or written by a single I/O request. Also, a set of related data processed as a unit when reading or writing a file.

Repeating Groups

An alternate-key attribute indicating that each data record can contain more than one value for the alternate key.

Rewind

For sequential and byte-addressable files, to position a file at its beginning of information (BOI). For keyed files, to position a file at the record with the lowest key value.

Ring

The level of hardware protection given a file or segment. A file is protected from unauthorized access by tasks executing in higher rings.

Ring_Attributes

A file attribute whose value consists of three ring numbers referenced as r1, r2, and r3. The ring numbers define four ring brackets for the file as follows:

Read bracket is 1 through r2.

Write bracket is 1 through r1.

Execute bracket is r1 through r2.

Call bracket is r2+1 through r3.

S

Sequential Access

An access mode in which records are processed in the order (physical or logical) in which they occur on a storage device. Contrast with Random Access.

Sequential File Organization

A file organization in which records can only be processed in physical order. Records are always read in the order that they were written to the file.

Sort

The process of arranging records in a specified order.

Sort Key

As used with the Sort/Merge interface, a field of information within each record in a sort or merge input file that is used to determine the order in which records are written to the output file.

Sort Order

Ordering of data according to key fields, either ascending or descending.

Source Code

Code written by the programmer in a language such as CYBIL, and input to a compiler.

Source Library

A collection of text units on a file, generated and manipulated by the Source Code Utility (SCU).

Sparse-Key Control

An alternate-key attribute that allows only certain records to be included in the alternate index. Inclusion or exclusion of a record is determined by the character at the sparse-key control position of the record.

Statistics

Counts maintained for a keyed file. Each type of file access is counted as well as the number of records in the file.

Status Variable

The variable in which the completion status of the command or procedure is returned.

Sum Fields

Used with the Sort/Merge interface, a record field containing a numeric value from the corresponding field of another record when the records are summed. The sum of the two values is stored in the new record that is created by the summing.

Summing

Used with the Sort/Merge interface, the process of combining two records having identical key values. The result of the process is a new record containing the original values of the key fields, the summed values of the sum fields, and data from one of the original records in any other record fields.

System Command Language (SCL)

The language that provides the interface to the features and capabilities of NOS/VE. All commands and statements are interpreted by SCL before being processed by the system.

T

Task

The instance of execution of a program.

U

U Record Type

Records for which the record structure is undefined.

Uncollated Key

A key consisting of 1 to 255 eight-bit characters. These keys are sorted by the magnitude of their binary ASCII code values. Contrast with Collated Key.

V

V Record Type

Variable-sized record; system default record type. Each V-type record has a record header. The header contains the record length and the length of the preceding record.

W

Working Storage Area

An area allocated by the task to hold data copied by get or put calls to a file.

ASCII Character Set B

This appendix lists the following character set:

ASCII (the only character set used by NOS/VE)

NOS/VE supports the American National Standards Institute (ANSI) ASCII character set (ANSI X3.4-1977). Although the ASCII character set contains 256 character codes, only the first 128 codes are used; the second 128 codes are unassigned. NOS/VE represents each 7-bit ASCII code in an 8-bit byte. The 7 bits are right-justified in each byte. For ASCII characters, the eighth or left-most bit is always zero.

Table B-1. ASCII Character Set

ASCII Code				
Decimal	**Hexadecimal**	**Octal**	**Graphic or Mnemonic**	**Name or Meaning**
000	00	000	NUL	Null
001	01	001	SOH	Start of heading
002	02	002	STX	Start of text
003	03	003	ETX	End of text
004	04	004	EOT	End of transmission
005	05	005	ENQ	Enquiry
006	06	006	ACK	Acknowledge
007	07	007	BEL	Bell
008	08	010	BS	Backspace
009	09	011	HT	Horizontal tabulation
010	0A	012	LF	Line feed
011	0B	013	VT	Vertical tabulation
012	0C	014	FF	Form feed
013	0D	015	CR	Carriage return
014	0E	016	SO	Shift out
015	0F	017	SI	Shift in
016	10	020	DLE	Data link escape
017	11	021	DC1	Device control 1
018	12	022	DC2	Device control 2
019	13	023	DC3	Device control 3
020	14	024	DC4	Device control 4
021	15	025	NAK	Negative acknowledge
022	16	026	SYN	Synchronous idle
023	17	027	ETB	End of transmission block
024	18	030	CAN	Cancel
025	19	031	EM	End of medium
026	1A	032	SUB	Substitute
027	1B	033	ESC	Escape
028	1C	034	FS	File separator
029	1D	035	GS	Group separator
030	1E	036	RS	Record separator
031	1F	037	US	Unit separator
032	20	040	SP	Space
033	21	041	!	Exclamation point
034	22	042	"	Quotation marks
035	23	043	#	Number sign
036	24	044	$	Dollar sign
037	25	045	%	Percent sign
038	26	046	&	Ampersand
039	27	047	'	Apostrophe
040	28	050	(	Opening parenthesis
041	29	051	)	Closing parenthesis
042	2A	052	*	Asterisk
043	2B	053	+	Plus

(Continued)

Table B-1. ASCII Character Set *(Continued)*

ASCII Code				
Decimal	**Hexadecimal**	**Octal**	**Graphic or Mnemonic**	**Name or Meaning**
044	2C	054	,	Comma
045	2D	055	-	Hyphen
046	2E	056	.	Period
047	2F	057	/	Slant
048	30	060	0	Zero
049	31	061	1	One
050	32	062	2	Two
051	33	063	3	Three
052	34	064	4	Four
053	35	065	5	Five
054	36	066	6	Six
055	37	067	7	Seven
056	38	070	8	Eight
057	39	071	9	Nine
058	3A	072	:	Colon
059	3B	073	;	Semicolon
060	3C	074	<	Less than
061	3D	075	=	Equals
062	3E	076	>	Greater than
063	3F	077	?	Question mark
064	40	100	@	Commercial at
065	41	101	A	Uppercase A
066	42	102	B	Uppercase B
067	43	103	C	Uppercase C
068	44	104	D	Uppercase D
069	45	105	E	Uppercase E
070	46	106	F	Uppercase F
071	47	107	G	Uppercase G
072	48	110	H	Uppercase H
073	49	111	I	Uppercase I
074	4A	112	J	Uppercase J
075	4B	113	K	Uppercase K
076	4C	114	L	Uppercase L
077	4D	115	M	Uppercase M
078	4E	116	N	Uppercase N
079	4F	117	O	Uppercase O
080	50	120	P	Uppercase P
081	51	121	Q	Uppercase Q
082	52	122	R	Uppercase R
083	53	123	S	Uppercase S
084	54	124	T	Uppercase T
085	55	125	U	Uppercase U
086	56	126	V	Uppercase V
087	57	127	W	Uppercase W
088	58	130	X	Uppercase X
089	59	131	Y	Uppercase Y

(Continued)

Table B-1. ASCII Character Set *(Continued)*

ASCII Code				
Decimal	**Hexadecimal**	**Octal**	**Graphic or Mnemonic**	**Name or Meaning**
090	5A	132	Z	Uppercase Z
091	5B	133	[	Opening bracket
092	5C	134	\	Reverse slant
093	5D	135	]	Closing bracket
094	5E	136	^	Circumflex
095	5F	137	_	Underline
096	60	140	`	Grave accent
097	61	141	a	Lowercase a
098	62	142	b	Lowercase b
099	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Opening brace
124	7C	174	|	Vertical line
125	7D	175	}	Closing brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

Constant and Type Declarations C

This appendix lists the constant and type declarations used by the procedures described in this manual. In general, the declarations are listed in alphabetical order by identifier name. However, the numeric order of ordinal constants is maintained.

AM

Constants

```
aac$access_method_ID = 'AA';
amc$access_method_id = 'AM';
amc$apl = 'APL',
amc$assembler = 'ASSEMBLER';
amc$basic = 'BASIC';
amc$cobol = 'COBOL';
amc$cybil = 'CYBIL';
amc$data = 'DATA';
amc$debugger = 'DEBUGGER';
amc$file_byte_limit = 4398046511103 { 2**42 - 1 }
{ bytes } ;
amc$fortran = 'FORTRAN';
amc$legible = 'LEGIBLE';
amc$library = 'LIBRARY';
amc$list = 'LIST';
amc$mau_length = 2048 { bytes } ;

amc$max_attribute = 511 { 01ff(16) } ;
amc$max_block_number = 0ffffffff(16);
amc$max_blocks_per_file = amc$file_byte_limit DIV
  amc$mau_length;
amc$max_buffer_length = 16777215 { 2**24 - 1 bytes } ;
amc$max_ecc_program_action = 161999;
amc$max_ecc_validation = 160999;
amc$max_error_count = 0ffff(16);
amc$max_fap_layers = 15;
amc$max_file_id_ordinal = 4095;
amc$max_home_blocks = amc$file_byte_limit;
amc$max_index_level = 15;
amc$max_info = 01ff(16);
amc$max_key_length = 255,
```

```
amc$max_key_position = 0ffff(16),
amc$max_label_length = osc$maximum_offset;
amc$max_line_number = 6;
amc$max_lines_per_inch = 12,
amc$max_operation = 01ff(16);
amc$max_optional_attributes = 72,
amc$max_page_width = 65535;
amc$max_path_name_size = 256;
amc$max_record_header = 16;
amc$max_records_per_block = 0ffff(16);
amc$max_statement_id_length = 17;
amc$max_tape_mark_count = 40000;
amc$max_user_info = 32;
amc$max_vol_number = 65536;

amc$maximum_block = 16777216 { 2**24 bytes } ;
amc$maximum_record = amc$file_byte_limit;

amc$min_ecc_program_action = 161000;
amc$min_ecc_validation = 160000;

amc$object = 'OBJECT';
amc$pascal = 'PASCAL';
amc$pli = 'PLI';
amc$ppu_assembler = 'PPU_ASSEMBLER';
amc$scl = 'SCL';
amc$scu = 'SCU';
amc$unknown_contents = 'UNKNOWN';
amc$unknown_processor = 'UNKNOWN';
amc$unknown_structure = 'UNKNOWN';
```

Ordinals

```
{}
{Codes 1..100 are reserved for operations which are}
{not passed to file_access_procedures.}
{}
    amc$access_method_req = 1,
    amc$add_to_file_description_req = 3,
    amc$allocate_req = 5,
    amc$change_file_attributes_cmd = 6,
    amc$compare_file_cmd = 7,
    amc$copy_file_cmd = 8,
    amc$copy_file_req = 9,
    amc$copy_partitions_req = 10,
    amc$copy_records_req = 11,
    amc$copy_partial_records_req = 12,
```

```
    amc$detach_file_cmd = 17,
    amc$display_file_attributes_cmd = 18,
    amc$display_file_cmd = 19,
    amc$evict_req = 20,
    amc$fetch_fap_pointer_req = 22,
    amc$file_req = 24,
    amc$get_file_attributes_req = 30,
    amc$label_req = 50,
    amc$override_file_attributes = 60,
    amc$rename_req = 72,
    amc$return_req = 74,
    amc$rewind_files_cmd = 75,
    amc$set_local_name_abnormal_req = 76,
    amc$set_file_attributes_cmd = 77,
    amc$set_file_inst_abnormal_req = 78,
    amc$skip_tape_marks_cmd = 81,
    amc$skip_tape_marks_req = 82,
    amc$store_fap_pointer_req = 84,
    amc$validate_caller_privilege = 95,
%}

{ Codes amc$fap_op_start..(amc$last_access_start-1) are }
{ reserved for operations which are passed to }
{ file_access_procedures but which are not recorded in }
{ last_access_operation status. }
%}
    amc$fap_op_start = 101,
    amc$fetch_access_information_rq = 101,
%}
{ Codes amc$last_access_start..amc$max_operation are }
{ reserved for operations which are passed to }
{ file_access_procedures. }
%}
    amc$last_access_start = 105,
    amc$check_buffer_req = 110,
    amc$check_record_req = 111,
    amc$close_req = 112,
    amc$close_volume_req = 113,
    amc$delete_req = 114,
    amc$delete_direct_req = 115,
    amc$delete_key_req = 116,
    amc$fetch_req = 117,
    amc$flush_req = 118,
    amc$get_direct_req = 119,
    amc$get_key_req = 120,
    amc$get_label_req = 121,
    amc$get_next_req = 122,
```

```
    amc$get_next_key_req = 123,
    amc$get_partial_req = 124,
    amc$get_segment_pointer_req = 126,
    amc$lock_file_req = 127,
    amc$lock_file = 127,
    amc$open_req = 128,
    amc$pack_block_req = 129,
    amc$pack_record_req = 130,
    amc$put_direct_req = 131,
    amc$put_key_req = 132,
    amc$put_label_req = 133,
    amc$put_next_req = 134,
    amc$put_partial_req = 135,
    amc$putrep_req = 137,
    amc$read_req = 138,
    amc$read_direct_req = 139,
    amc$read_direct_skip_req = 140,
    amc$read_skip_req = 141,
    amc$replace_req = 142,
    amc$replace_direct_req = 143,
    amc$replace_key_req = 144,
    amc$rewind_req = 145,
    amc$rewind_volume_req = 146,
    amc$seek_direct_req = 147,
    amc$set_segment_eoi_req = 148,
    amc$set_segment_position_req = 149,
    amc$skip_req = 150,
    amc$start_req = 151,
    amc$store_req = 152,
    amc$unlock_file_req = 153,
    amc$unlock_file = 153,
    amc$unpack_block_req = 154,
    amc$unpack_record_req = 155,
    amc$write_req = 156,
    amc$write_direct_req = 157,
    amc$write_end_partition_req = 158,
    amc$write_tape_mark_req = 159,
    ifc$fetch_terminal_req = 160,
    ifc$store_terminal_req = 161,
    amc$abandon_key_definitions = 162,
    amc$abort_file_parcel = 163,
    amc$apply_key_definitions = 164,
    amc$begin_file_parcel = 165,
    amc$check_nowait_request = 166,
    amc$commit_file_parcel = 167,
    amc$create_key_definition = 168,
    amc$create_nested_file = 169,
```

```
    amc$delete_key_definition = 170,
    amc$delete_nested_file = 171,
    amc$find_record_space = 172,
    amc$get_key_definitions = 173,
    amc$get_lock_keyed_record = 174,
    amc$get_lock_next_keyed_record = 175,
    amc$get_nested_file_definitions = 176,
    amc$get_next_primary_key_list = 177,
    amc$get_primary_key_count = 178,
    amc$get_space_used_for_key = 179,
    amc$lock_key = 180,
    amc$select_key = 181,
    amc$select_nested_file = 182,
    amc$separate_key_groups = 183,
    amc$unlock_key = 184,
%}
    amc$block_number = 1,
    amc$current_byte_address = 2,
    amc$eoi_byte_address = 3,
    amc$error_count = 4 { Supported only for       }
                        { indexed_sequential files },
    amc$error_status = 5,
    amc$file_position = 6,
    amc$last_access_operation = 7,
    amc$last_op_status = 8,
    amc$levels_of_indexing = 9 { Supported only for       }
                               { indexed_sequential files },
    amc$previous_record_address = 10,
    amc$previous_record_length = 11,
    amc$residual_skip_count = 12,
    amc$volume_position = 13,
    amc$volume_number = 14,
    amc$duplicate_value_inserted = 15,
    amc$number_of_nested_files = 16,
    amc$null_item = 17,
    amc$number_of_volumes = 18,
    amc$primary_key = 19,
    amc$selected_key_name = 20,
    amc$selected_nested_file = 21,
    amc$volume_description = 22,
%}
    amc$access_level = 1,
    amc$access_mode = 2,
    amc$application_info = 3,
    amc$average_record_length = 4,
    amc$block_type = 5,
    amc$character_conversion = 6,
```

```
    amc$clear_space = 7,
    amc$collate_table = 8,
    amc$collate_table_name = 9,
    amc$data_padding = 12,
    amc$embedded_key = 13,
    amc$error_exit_name = 14,
    amc$error_exit_procedure = 15,
    amc$error_limit = 16,
    amc$error_options = 17,
    amc$estimated_record_count = 18,
    amc$file_access_procedure = 19,
    amc$file_contents = 20,
    amc$file_length = 21,
    amc$file_limit = 22,
    amc$file_organization = 24,
    amc$file_processor = 25,
    amc$file_structure = 26,
    amc$forced_write = 27,
    amc$global_access_mode = 28,
    amc$global_file_address = 29,
    amc$global_file_position = 30,
    amc$global_file_name = 31,
    amc$global_share_mode = 32,
    amc$index_levels = 33,
    amc$index_padding = 34,
    amc$internal_code = 35,
    amc$key_length = 36,
    amc$key_position = 37,
    amc$key_type = 38,
    amc$label_exit_name = 39,
    amc$label_exit_procedure = 40,
    amc$label_options = 41,
    amc$label_type = 42,
    amc$line_number = 44,
    amc$max_block_length = 45,
    amc$max_record_length = 46,
    amc$message_control = 47,
    amc$min_block_length = 48,
    amc$min_record_length = 49,
    amc$null_attribute = 50,
    amc$open_position = 51,
    amc$padding_character = 52,
    amc$page_format = 53,
    amc$page_length = 54,
    amc$page_width = 55,
    amc$permanent_file = 56,
    amc$preset_value = 57,
```

```
    amc$record_limit = 59,
    amc$record_type = 60,
    amc$records_per_block = 61,
    amc$return_option = 62,
    amc$ring_attributes = 63,
    amc$statement_identifier = 64,
    amc$user_info = 66,
    amc$vertical_print_density = 67,
    amc$compression_procedure_name = 68,
    amc$dynamic_home_block_space = 69,
    amc$hashing_procedure_name = 70,
    amc$initial_home_block_count = 71,
    amc$loading_factor = 72,
    amc$lock_expiration_time = 73,
    amc$logging_options = 74,
    amc$log_residence = 75,
{}
    amc$concatenated_key_portion = 100,
    amc$duplicate_keys = 101,
    amc$group_name = 102,
    amc$null_suppression = 103,
    amc$repeating_group = 104,
    amc$sparse_keys = 105,
    amc$variable_length_key = 106,
```

Types

```
amt$access_info = record
  item_returned {output} : boolean,
  case key { input } : amt$access_info_keys of
{ output }
  = amc$block_number =
    block_number: amt$block_number,
  = amc$current_byte_address =
    current_byte_address: amt$file_byte_address,
  = amc$duplicate_value_inserted =
    duplicate_value_inserted: boolean,
  = amc$eoi_byte_address =
    eoi_byte_address: amt$file_byte_address,
  = amc$error_count =
    error_count: amt$error_count,
  = amc$error_status =
    error_status: ost$status_condition,
  = amc$file_position =
    file_position: amt$file_position,
  = amc$last_access_operation =
    last_access_operation:
      amt$last_access_operation,
  = amc$last_op_status =
    last_op_status: amt$last_op_status,
  = amc$levels_of_indexing =
    levels_of_indexing: amt$index_levels,
  = amc$null_item =
    ,
  = amc$number_of_nested_files =
    number_of_nested_files: amt$nested_file_count,
  = amc$number_of_volumes =
    number_of_volumes: amt$volume_number,
  = amc$previous_record_address =
    previous_record_address: amt$file_byte_address,
  = amc$previous_record_length =
    previous_record_length: amt$max_record_length,
  = amc$primary_key =
    primary_key: amt$primary_key,
  = amc$residual_skip_count =
    residual_skip_count: amt$residual_skip_count,
  = amc$selected_key_name =
    selected_key_name: amt$selected_key_name,
  = amc$selected_nested_file =
    selected_nested_file: amt$selected_nested_file,
```

```
  = amc$volume_description =
    volume_index {input} : amt$volume_number,
    volume_description {output} : rmt$volume_descriptor,
  = amc$volume_number =
    volume_number: amt$volume_number,
  = amc$volume_position =
    volume_position: amt$volume_position,
  casend
recend;

amt$access_info_keys = 1 .. amc$max_info;

amt$access_information = array [1 .. * ] of
  amt$access_info;

amt$access_level = (amc$physical, amc$record,
  amc$segment);

amt$access_selection = amt$file_item;

amt$attribute_source = (amc$undefined_attribute,
  amc$local_file_information,
  amc$change_file_attributes, amc$open file_request,
  amc$file_reference, amc$file_command,
  amc$file_request, amc$add_to_file_description,
  amc$access_method_default, amc$store_request)

amt$average_record_length = 1 .. amc$maximum_record;

amt$basic_key_definition = record
  case definition_returned: boolean of
  = TRUE =
    key_name: amt$key_name,
    key_position: amt$key_position,
    key_length: amt$key_length,
    number_of_optional_attributes: amt$max_optional_attributes,
  casend,
recend,

amt$begin_file_parcel = record
  general_commit: amt$general_commit,
recend;

 amt$block_header_type = (amc$tapemark_block,
  amc$data_block);
```

```
amt$block_number = 1 .. amc$max_block_number;

amt$block_status = (amc$no_error,
  amc$recovered_error, amc$unrecovered_error);

amt$block_type = (amc$system_specified,
  amc$user_specified);

amt$buffer_area = ^SEQ ( * );

amt$buffer_length = amc$mau_length ..
  amc$max_buffer_length;

amt$collate_table = array [char] of
  amt$collation_value;

amt$collation_value = 0 .. 255;

amt$commit_file_parcel = record
  phase: amt$commit_phase,
recend;

amt$commit_phase = (amc$simple_commit, amc$tentative_commit,
  amc$permanent_commit);

amt$compression_effect = (amc$compress, amc$decompress);

amt$compression_procedure = ^procedure
  (effect: amt$compression_effect;
  input_working_storage_area: ^cell;
  input_working_storage_length: amt$max_record_length;
  output_working_storage_area: ^cell;
  key_position: amt$key_position;
  key_length: amt$key_length;
  VAR output_working_storage_length: amt$max_record_length;
  VAR record_left_uncompressed: boolean;
  VAR status: ost$status);

amt$compression_procedure_name = amt$entry_point_reference;

amt$create_key_definition = record
  key_name: amt$key_name,
  key_position: amt$key_position,
  key_length: amt$key_length,
  optional_attributes: ^amt$optional_key_attributes,
recend;
```

```
amt$create_nested_file = record
  definition: ^amt$nested_file_definition,
recend;

amt$creation_date = 1 .. 99999 { yyddd, defaults }
  { to current date };

amt$data_block_count = 1 .. amc$max_blocks_per_file;

amt$data_padding = 0 .. 99 { expressed as a }
  { percentage } ;

amt$delete_key_definition = record
  key_name: amt$key_name,
recend;

amt$delete_nested_file = record
  nested_file_name: amt$nested_file_name,
recend;

amt$duplicate_key_control = (amc$no_duplicates_allowed,
  amc$first_in_first_out, amc$ordered_by_primary_key);

amt$duplicate_value_inserted = boolean;

amt$dynamic_home_block_space = boolean;

amt$error_count = 0 .. amc$max_error_count;

amt$entry_point_reference = record
  name: pmt$program_name,
  object_library: amt$path_name,
recend;

amt$error_exit_procedure = ^procedure
  (file_identifier: amt$file_identifier;
  VAR status: ost$status);

amt$error_limit = 0 .. 0ffff(16);

amt$error_options = (amc$terminate_file,
  amc$drop_block, amc$accept_record);

amt$estimated_record_count = integer;

amt$expiration_date = 1 .. 99999 { yyddd, defaults }
{ to creation date plus one year };
```

```
amt$fap_layer_number = 0 .. amc$max_fap_layers;

amt$fetch_attributes = array [1 .. * ] of
  amt$fetch_item;

amt$fetch_item = record
  source { output } : amt$attribute_source,
  case key { input } : amt$file_attribute_keys of
{ output }
  = amc$access_level =
    access_level: amt$access_level,
  = amc$access_mode =
    access_mode: pft$usage_selections,
  = amc$application_info =
    application_info: pft$application_info,
  = amc$block_type =
    block_type: amt$block_type,
  = amc$character_conversion =
    character_conversion: boolean,
  = amc$clear_space =
    clear_space: ost$clear_file_space,
  = amc$error_exit_name =
    error_exit_name: pmt$program_name,
  = amc$error_exit_procedure =
    error_exit_procedure: amt$error_exit_procedure,
  = amc$error_options =
    error_options: amt$error_options,
  = amc$file_access_procedure =
    file_access_procedure: pmt$program_name,
  = amc$file_contents =
    file_contents: amt$file_contents,
  = amc$file_limit =
    file_limit: amt$file_limit,
  = amc$file_organization =
    file_organization: amt$file_organization,
  = amc$file_processor =
    file_processor: amt$file_processor,
  = amc$file_structure =
    file_structure: amt$file_structure,
  = amc$forced_write =
    forced_write: amt$forced_write,
  = amc$global_access_mode =
    global_access_mode: pft$usage_selections,
  = amc$global_file_address =
    global_file_address: amt$file_byte_address,
  = amc$global_file_name =
    global_file_name: ost$binary_unique_name,
```

```
= amc$global_file_position =
  global_file_position: amt$global_file_position,
= amc$global_share_mode =
  global_share_mode: pft$share_selections,
= amc$internal_code =
  internal_code: amt$internal_code,
= amc$label_exit_name =
  label_exit_name: pmt$program_name,
= amc$label_exit_procedure =
  label_exit_procedure: amt$label_exit_procedure,
= amc$label_options =
  label_options: amt$label_options,
= amc$label_type =
  label_type: amt$label_type,
= amc$line_number =
  line_number: amt$line_number,
= amc$max_block_length =
  max_block_length: amt$max_block_length,
= amc$max_record_length =
  max_record_length: amt$max_record_length,
= amc$min_block_length =
  min_block_length: amt$min_block_length,
= amc$min_record_length =
  min_record_length: amt$min_record_length,
= amc$null_attribute =
  ,
= amc$open_position =
  open_position: amt$open_position,
= amc$padding_character =
  padding_character: amt$padding_character,
= amc$page_format =
  page_format: amt$page_format,
= amc$page_length =
  page_length: amt$page_length,
= amc$page_width =
  page_width: amt$page_width,
= amc$permanent_file =
  permanent_file: boolean,
= amc$preset_value =
  preset_value: amt$preset_value,
= amc$record_type =
  record_type: amt$record_type,
= amc$ring_attributes =
  ring_attributes: amt$ring_attributes,
= amc$statement_identifier =
  statement_identifier: amt$statement_identifier,
```

```
  = amc$user_info =
    user_info: amt$user_info,
  = amc$average_record_length =
    average_record_length:
      amt$average_record_length,
  = amc$collate_table =
    collate_table: ^amt$collate_table,
  = amc$collate_table_name =
    collate_table_name: pmt$program_name,
  = amc$compression_procedure_name =
    compression_procedure_name: [input,output]
      ^amt$compression_procedure_name,
  = amc$data_padding =
    data_padding: amt$data_padding,
  = amc$dynamic_home_block_space =
    dynamic_home_block_space:
      amt$dynamic_home_block_space,
  = amc$embedded_key =
    embedded_key: boolean,
  = amc$error_limit =
    error_limit: amt$error_limit,
  = amc$estimated_record_count =
    estimated_record_count:
      amt$estimated_record_count,
  = amc$hashing_procedure_name =
    hashing_procedure_name: [input,output]
      ^amt$hashing_procedure_name,
  = amc$index_levels =
    index_levels: amt$index_levels,
  = amc$index_padding =
    index_padding: amt$index_padding,
  = amc$initial_home_block_count =
    initial_home_block_count:
      amt$initial_home_block_count,
  = amc$key_length =
    key_length: amt$key_length,
  = amc$key_position =
    key_position: amt$key_position,
  = amc$key_type =
    key_type: amt$key_type,
  = amc$loading_factor =
    loading_factor: amt$loading_factor,
  = amc$lock_expiration_time =
    lock_expiration_time: amt$lock_expiration_time,
  = amc$logging_options =
    logging_options: amt$logging_options,
```

```
  = amc$log_residence =
    log_residence: [input,output]
      ^amt$log_residence,
  = amc$message_control =
    message_control: amt$message_control,
  = amc$record_limit =
    record_limit: amt$record_limit,
  = amc$records_per_block =
    records_per_block: amt$records_per_block,
  casend,
recend;

amt$file_access_code = char { defaults to space },

amt$file_access_selections = ^array [1 .. * ] of
  amt$access_selection,

amt$file_attribute_keys = 1 .. amc$max_attribute,

amt$file_attributes = array [1 .. * ] of
  amt$file_item,

amt$file_byte_address = 0 .. amc$file_byte_limit;

amt$file_contents = ost$name;

amt$file_id_ordinal = 0 .. amc$max_file_id_ordinal,

amt$file_id_sequence = 1 .. 4095;

amt$file_identifier = record
  ordinal: amt$file_id_ordinal,
  sequence: amt$file_id_sequence,
recend,

amt$file_item = record
  case key { input } : amt$file_attribute_keys of
  { input }
  = amc$access_mode =
    access_mode: pft$usage_selections,
  = amc$block_type =
    block_type: amt$block_type,
  = amc$character_conversion =
    character_conversion: boolean,
  = amc$clear_space =
    clear_space: ost$clear_file_space,
```

```
  = amc$error_exit_name =
    error_exit_name: pmt$program_name,
  = amc$error_options =
    error_options: amt$error_options,
  = amc$file_access_procedure =
    file_access_procedure: pmt$program_name,
  = amc$file_contents =
    file_contents: amt$file_contents,
  = amc$file_limit =
    file_limit: amt$file_limit,
  = amc$file_organization =
    file_organization: amt$file_organization,
  = amc$file_processor =
    file_processor: amt$file_processor,
  = amc$file_structure =
    file_structure: amt$file_structure,
  = amc$forced_write =
    forced_write: amt$forced_write,
  = amc$internal_code =
    internal_code: amt$internal_code,
  = amc$label_exit_name =
    label_exit_name: pmt$program_name,
  = amc$label_options =
    label_options: amt$label_options,
  = amc$label_type =
    label_type: amt$label_type,
  = amc$line_number =
    line_number: amt$line_number,
  = amc$max_block_length =
    max_block_length: amt$max_block_length,
  = amc$max_record_length =
    max_record_length: amt$max_record_length,
  = amc$min_block_length =
    min_block_length: amt$min_block_length,
  = amc$min_record_length =
    min_record_length: amt$min_record_length,
  = amc$null_attribute =
    ,
  = amc$open_position =
    open_position: amt$open_position,
  = amc$padding_character =
    padding_character: amt$padding_character,
  = amc$page_format =
    page_format: amt$page_format,
  = amc$page_length =
    page_length: amt$page_length,
```

```
= amc$page_width =
  page_width: amt$page_width,
= amc$preset_value =
  preset_value: amt$preset_value,
= amc$record_type =
  record_type: amt$record_type,
= amc$return_option =
  return_option: amt$return_option,
= amc$ring_attributes =
  ring_attributes: amt$ring_attributes,
= amc$statement_identifier =
  statement_identifier: amt$statement_identifier,
= amc$user_info =
  user_info: amt$user_info,
= amc$vertical_print_density =
  vertical_print_density:
    amt$vertical_print_density,
= amc$average_record_length =
  average_record_length:
    amt$average_record_length,
= amc$collate_table_name =
  collate_table_name: pmt$program_name,
= amc$compression_procedure_name =
  compression_procedure_name: [input,output]
    ^amt$compression_procedure_name,
= amc$data_padding =
  data_padding: amt$data_padding,
= amc$dynamic_home_block_space =
  dynamic_home_block_space: amt$dynamic_home_block_space,
= amc$embedded_key =
  embedded_key: boolean,
= amc$error_limit =
  error_limit: amt$error_limit,
= amc$estimated_record_count =
  estimated_record_count:
    amt$estimated_record_count,
= amc$hashing_procedure_name =
  hashing_procedure_name: [input,output]
    ^amt$hashing_procedure_name,
= amc$index_levels =
  index_levels: amt$index_levels,
= amc$index_padding =
  index_padding: amt$index_padding,
= amc$initial_home_block_count =
  initial_home_block_count: amt$initial_home_block_count,
= amc$key_length =
  key_length: amt$key_length,
```

```
  = amc$key_position =
    key_position: amt$key_position,
  = amc$key_type =
    key_type: amt$key_type,
  = amc$loading_factor =
    loading_factor: amt$loading_factor,
  = amc$lock_expiration_time =
    lock_expiration_time: amt$lock_expiration_time,
  = amc$logging_options =
    logging_options: amt$logging_options,
  = amc$log_residence =
    log_residence: {input,output}
      ^amt$log_residence,
  = amc$message_control =
    message_control: amt$message_control,
  = amc$record_limit =
    record_limit: amt$record_limit,
  = amc$records_per_block =
    records_per_block: amt$records_per_block,
  casend
recend;

amt$file_length = 0 .. amc$file_byte_limit;

amt$file_limit = 0 .. amc$file_byte_limit;

amt$file_lock = (amc$lock_set, amc$already_set);

amt$file_organization = (amc$sequential, amc$byte_addressable,
  amc$indexed_sequential, amc$direct_access, amc$system_key);

 amt$file_position = (amc$boi, amc$bop,
  amc$mid_record, amc$eor, amc$eop, amc$eoi, amc$end_of_key_list);

amt$file_processor = ost$name;

amt$file_reference = string ( * <= amc$max_path_name_size);

amt$file_set_id = string (6), { defaults to spaces };

amt$file_structure = ost$name;

amt$find_record_space = record
  space: amt$file_length,
  where: amt$put_locality,
  wait: ost$wait,
recend;
```

```
amt$forced_write = (amc$forced,
  amc$forced_if_structure_change, amc$unforced),

amt$general_commit = record
  case general_commit_in_use: boolean of
  = TRUE =
    general_commit_name: ost$name,
  casend,
recend;

amt$generation_number = 1 .. 9999 { defaults }
{ to 0001 };
amt$get_attributes = array [1 .. * ] of
  amt$get_item;

amt$get_item = record
  source { output }: amc$undefined_attribute ..
    amc$access_method_default,
  case key { input } : amt$file_attribute_keys of
{ output }
  = amc$access_mode =
    access_mode: pft$usage_selections,
  = amc$application_info =
    application_info: pft$application_info,
  = amc$block_type =
    block_type: amt$block_type,

  = amc$character_conversion =
    character_conversion: boolean,
  = amc$clear_space =
    clear_space: ost$clear_file_space,
  = amc$error_exit_name =
    error_exit_name: pmt$program_name,
  = amc$error_options =
    error_options: amt$error_options,
  = amc$file_access_procedure =
    file_access_procedure: pmt$program_name,
  = amc$file_contents =
    file_contents: amt$file_contents,
  = amc$file_length =
    file_length: amt$file_length,
  = amc$file_limit =
    file_limit: amt$file_limit,
  = amc$file_organization =
    file_organization: amt$file_organization,
  = amc$file_processor =
    file_processor: amt$file_processor,
```

```
  = amc$file_structure =
    file_structure: amt$file_structure,
  = amc$forced_write =
    forced_write: amt$forced_write,
  = amc$global_access_mode =
    global_access_mode: pft$usage_selections,
  = amc$global_file_address =
    global_file_address: amt$file_byte_address,
  = amc$global_file_name =
    global_file_name: ost$binary_unique_name,
  = amc$global_file_position =
    global_file_position: amt$global_file_position,
  = amc$global_share_mode =
    global_share_mode: pft$share_selections,
  = amc$internal_code =
    internal_code: amt$internal_code,
  = amc$label_exit_name =
    label_exit_name: pmt$program_name,
  = amc$label_options =
    label_options: amt$label_options,
  = amc$label_type =
    label_type: amt$label_type,
  = amc$line_number =
    line_number: amt$line_number,
  = amc$max_block_length =
    max_block_length: amt$max_block_length,
  = amc$max_record_length =
    max_record_length: amt$max_record_length,
  = amc$min_block_length =
    min_block_length: amt$min_block_length,
  = amc$min_record_length =
    min_record_length: amt$min_record_length,
  = amc$null_attribute =
    ,
  = amc$open_position =
    open_position: amt$open_position,
  = amc$padding_character =
    padding_character: amt$padding_character,

  = amc$page_format =
    page_format: amt$page_format,
  = amc$page_length =
    page_length: amt$page_length,
  = amc$page_width =
    page_width: amt$page_width,
  = amc$permanent_file =
    permanent_file: boolean,
```

```
= amc$preset_value =
  preset_value: amt$preset_value,
= amc$record_type =
  record_type: amt$record_type,
= amc$return_option =
  return_option: amt$return_option,
= amc$ring_attributes =
  ring_attributes: amt$ring_attributes,
= amc$statement_identifier =
  statement_identifier: amt$statement_identifier,
= amc$user_info =
  user_info: amt$user_info,
= amc$vertical_print_density =
  vertical_print_density:
    amt$vertical_print_density,
= amc$average_record_length =
  average_record_length:
    amt$average_record_length,
= amc$collate_table_name =
  collate_table_name: pmt$program_name,
= amc$compression_procedure_name =
  compression_procedure_name: {input,output}
    ^amt$compression_procedure_name,
= amc$data_padding =
  data_padding: amt$data_padding,
= amc$dynamic_home_block_space =
  dynamic_home_block_space:
    amt$dynamic_home_block_space
= amc$embedded_key =
  embedded_key: boolean,
= amc$error_limit =
  error_limit: amt$error_limit,
= amc$estimated_record_count =
  estimated_record_count:
    amt$estimated_record_count,
= amc$hashing_procedure_name =
    hashing_procedure_name: {input,output}
      ^amt$hashing_procedure_name,
= amc$index_levels =
  index_levels: amt$index_levels,
= amc$index_padding =
  index_padding: amt$index_padding,
= amc$key_length =
  key_length: amt$key_length,
= amc$key_position =
  key_position: amt$key_position,
```

```
  = amc$key_type =
    key_type: amt$key_type,
  = amc$loading_factor =
    loading_factor: amt$loading_factor,
  = amc$lock_expiration_time =
    lock_expiration_time: amt$lock_expiration_time,
  = amc$logging_options =
    logging_options: amt$logging_options
  = amc$log_residence =
    log_residence: {input,output}
      ^amt$log_residence
  = amc$message_control =
    message_control: amt$message_control,
  = amc$record_limit =
    record_limit: amt$record_limit,
  = amc$records_per_block =
    records_per_block: amt$records_per_block,
  casend
recend;

amt$get_key_definitions = record
  key_definitions: ^SEQ (*),
recend;

amt$get_lock_keyed_record = record
  working_storage_area: ^cell,
  working_storage_length: amt$working_storage_length,
  key_location: ^cell,
  major_key_length: amt$major_key_length,
  relation: amt$key_relation,
  wait_for_lock: ost$wait_for_lock,
  unlock_control: amt$unlock_control,
  lock_intent: amt$lock_intent,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;
```

```
amt$get_lock_next_keyed_record = record
  working_storage_area: ^cell,
  working_storage_length: amt$working_storage_length,
  key_location: ^cell,
  wait_for_lock: ost$wait_for_lock,
  unlock_control: amt$unlock_control,
  lock_intent: amt$lock_intent,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

amt$get_nested_file_definitions = record
  definitions: ^amt$nested_file_definitions,
  nested_file_count: ^amt$nested_file_count,
recend;

amt$get_next_primary_key_list = record
  high_key: ^cell,
  major_high_key: amt$major_key_length,
  high_key_relation: amt$key_relation,
  working_storage_area: ^cell,
  working_storage_length: amt$working_storage_length,
  end_of_primary_key_list: ^boolean,
  transferred_byte_count: ^amt$working_storage_length,
  transferred_key_count: ^amt$key_count_limit,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;
```

```
amt$get_primary_key_count = record
  low_key: ^cell,
  major_low_key: amt$major_key_length,
  low_key_relation: amt$key_relation,
  high_key: ^cell,
  major_high_key: amt$major_key_length,
  high_key_relation: amt$key_relation,
  list_count_limit: amt$key_count_limit,
  list_count: ^amt$key_count_limit,
  wait: ost$wait,
recend;

amt$global_file_position = amt$file_position;

amt$group_name = amt$key_name;

amt$hashing_procedure = ^procedure (old_key: ^cell;
  key_length: amt$key_length;
  VAR hashed_key: integer;
  VAR status: ost$status);

amt$hashing_procedure_name = amt$entry_point_reference;

amt$index_levels = 0 .. amc$max_index_level;

amt$index_padding = 0 .. 99 { expressed as a }
{ percentage };

amt$initial_home_block_count = 1 .. amc$max_home_blocks;

amt$internal_code = (amc$as6, amc$as8, amc$ascii,
  amc$d64, amc$ebcdic, amc$bcd);

amt$key_count_limit = 0 .. amc$file_byte_limit;

amt$key_length = 1 .. amc$max_key_length;

amt$key_name = ost$name;

amt$key_position = 0 .. amc$max_key_position;

amt$key_relation = (amc$equal_key,
  amc$greater_or_equal_key, amc$greater_key);

amt$key_type = (amc$collated_key, amc$integer_key,
  amc$uncollated_key);
```

```
amt$label_area_length = 18 .. amc$max_label_length;

amt$label_exit_procedure = ^procedure
  (file_identifier: amt$file_identifier);

amt$label_options = set of (amc$vol1, amc$uvl,
  amc$hdr1, amc$hdr2, amc$eov1, amc$eov2, amc$uhl,
  amc$eof1, amc$eof2, amc$utl);

amt$label_type = (amc$labelled,
  amc$non_standard_labelled, amc$unlabelled);

amt$last_access_operation = amc$last_access_start ..
  amc$max_operation;

amt$last_op_status = (amc$active, amc$complete);

amt$last_operation = 1 .. amc$max_operation;

amt$line_number = record
  length: amt$line_number_length,
  location: amt$line_number_location,
recend;

amt$line_number_length = 1 .. amc$max_line_number;

amt$line_number_location = amt$page_width;

amt$loading_factor = 0 .. 100;

amt$local_file_name = ost$name;

amt$lock_expiration_time = 0 .. 604800000 { milliseconds } ;

amt$lock_intent = (amc$exclusive_access, amc$preserve_access_
     and_content,amc$preserve_content);

amt$lock_file = record
  wait_for_lock: ost$wait_for_lock,
  lock_intent: amt$lock_intent,
recend;

amt$lock_key = record
  key_location: ^cell,
  wait_for_lock: ost$wait_for_lock,
  unlock_control: amt$unlock_control,
  lock_intent: amt$lock_intent,
recend;
```

```
amt$logging_options = set of amt$logging_possibilities;

amt$logging_possibilities = (amc$enable_parcels, amc$enable_
      media_recovery,amc$enable_request_recovery);

amt$log_residence = amt$path_name;

amt$major_key_length = 0 .. amc$max_key_length;

amt$max_block_length = 1 .. amc$maximum_block - 1;

amt$max_optional_attributes = 1 .. amc$max_optional_attributes;

amt$max_record_length = 0 .. amc$maximum_record;

amt$max_repeating_group_count = amt$max_record_length;

amt$message_control = set of (amc$trivial_errors,
  amc$messages, amc$statistics);

amt$min_block_length = 18 .. amc$maximum_block;

amt$min_record_length = 0 .. amc$maximum_record;

amt$nested_file_count = 1 .. amc$max_blocks_per_file;

amt$nested_file_definition = record
  nested_file_name: amt$nested_file_name,
  embedded_key: boolean,
  key_position: amt$key_position,
  key_length: amt$key_length,
  maximum_record: amt$max_record_length,
  minimum_record: amt$min_record_length,
  record_type: amt$record_type,
  case file_organization: amt$file_organization of
  = amc$indexed_sequential =
    key_type: amt$key_type,
    collate_table_name: pmt$program_name,
    data_padding: amt$data_padding,
    index_padding: amt$index_padding,
  = amc$direct_access =
    home_block_count: amt$initial_home_block_count,
    dynamic_home_block_space: amt$dynamic_home_block_space,
    loading_factor: amt$loading_factor,
    hashing_procedure: amt$hashing_procedure_name,
  = amc$system_key =
    records_per_block: amt$records_per_block,
  casend,
recend;
```

```
amt$nested_file_definitions = array [1 .. * ] of
  amt$nested_file_definition;

amt$nested_file_name = ost$name;

amt$nowait_var_parameters = SEQ (REP 10 of integer);

amt$open_position = (amc$open_no_positioning,
  amc$open_at_boi, amc$open_at_bop, amc$open_at_eoi);

amt$optional_key_attribute = record
  case selector: amt$file_attribute_keys of
  = amc$key_type =
    key_type: amt$key_type,
  = amc$collate_table_name =
    collate_table_name: pmt$program_name,
  = amc$duplicate_keys =
    duplicate_key_control: amt$duplicate_key_control,
  = amc$null_suppression =
    null_suppression: boolean,
  = amc$sparse_keys =
    sparse_key_control_position: amt$key_position,
    sparse_key_control_characters: set of char,
    sparse_key_control_effect: amt$sparse_key_control_effect,
  = amc$repeating_group =
    repeating_group_length: amt$max_record_length,
    repetition_control: amt$repetition_control,
  = amc$concatenated_key_portion =
    concatenated_key_position: amt$key_position,
    concatenated_key_length: amt$key_length,
    concatenated_key_type: amt$key_type,
  = amc$group_name =
    group_name: amt$group_name,
  = amc$variable_length_key =
    key_delimiter_characters: set of char,
  casend,
recend;

amt$optional_key_attributes = array [1 .. * ] of
  amt$optional_key_attribute;
```

```
amt$pack_block_header = record
  header_type: amt$block_header_type,
  block_length: amt$max_block_length,
  block_number: amt$block_number,
  unused_bit_count: amt$unused_bit_count,
recend;

amt$padding_character = char;

amt$page_format = (amc$continuous_form,
  amc$burstable_form, amc$non_burstable_form);

amt$page_length = 1 .. amc$file_byte_limit,

amt$page_width = 1 .. amc$max_page_width;

amt$path_name = string (amc$max_path_name_size);

amt$physical_transfer_count = 0 ..
  amc$max_buffer_length;

amt$pointer_kind = (amc$cell_pointer,
  amc$heap_pointer, amc$sequence_pointer);

amt$preset_value = integer;

amt$primary_key = ^cell;

amt$put_locality = (amc$put_near_anywhere, amc$put_near_get,
  amc$put_near_update);

amt$record_header = record
  header_type: amt$record_header_type,
  length: amt$max_record_length,
  previous_length: amt$max_record_length,
  unused_bit_count: amt$unused_bit_count,
  user_information: cell,
recend;

amt$record_header_length = 0 ..
  amc$max_record_header;

amt$record_header_type = (amc$full_record,
  amc$start_record, amc$continued_record,
  amc$end_record, amc$partition,
  amc$deleted_record);
```

```
amt$record_limit = 1 .. amc$file_byte_limit;

amt$record_type = (amc$variable { V } ,
  amc$undefined { U } , amc$ansi_fixed { F } ,
  amc$ansi_spanned { S } , amc$ansi_variable { D } );

amt$records_per_block = 1 .. amc$max_records_per_block;

amt$recovered_request = record
  past_last: boolean,
  task_id: pmt$task_id,
  file_identifier: amt$file_identifier,
  nested_file_selection: amt$nested_file_name,
  call_block: amt$call_block,
  status: ost$status,
  working_storage_length: amt$working_storage_length,
  key_length: amt$key_length,
recend;

amt$recovery_description = record
  case recover_option: amt$recovery_options of
  = amc$recover_file_media =
    media_recovery: record
      backup_date_time: ost$date_time,
      last_requests: ^SEQ ( * ),
    recend,
  = amc$recover_to_last_requests =
    last_requests: ^SEQ ( * ),
  = amc$recover_file_structure =
    ,
  = amc$salvage_data_records =
    new_keyed_file: amt$local_file_name,
    salvage_log: amt$salvage_log_description,
  casend,
recend;

amt$recovery_options = (amc$recover_file_media,
  amc$recover_to_last_requests, amc$recover_file_structure,
  amc$salvage_data_records);

amt$repetition_control = record
  case repeat_to_end_of_record: boolean of
  = FALSE =
    repeating_group_count: amt$max_repeating_group_count,
  casend,
recend;
```

```
amt$residual_skip_count = amt$skip_count;

amt$return_option = (amc$return_at_close,
  amc$return_at_task_exit, amc$return_at_job_exit);

amt$ring_attributes = record
  r1: ost$valid_ring,
  r2: ost$valid_ring,
  r3: ost$valid_ring,
recend;

amt$salvage_log_description = record
  case salvage_log_wanted: boolean of
  = TRUE =
    rejects_file: amt$local_file_name,
  casend,
recend;

amt$section_number = 1 .. 9999,
{ defaults to 0001 };

amt$segment_pointer = record
  case kind: amt$pointer_kind of
  =amc$cell_pointer=
    cell_pointer: ^cell,
  =amc$heap_pointer=
    heap_pointer: ^HEAP (*),
  =amc$sequence_pointer=
    sequence_pointer: ^SEQ (*),
  casend,
recend;

amt$select_key = record
  key_name: amt$key_name,
recend;

amt$select_nested_file = record
  nested_file_name: amt$nested_file_name,
recend;

amt$separate_key_groups = record
  group: amt$group_name,
  parallel_group: amt$group_name,
recend;

amt$selected_key_name = amt$key_name;

amt$selected_nested_file = amt$nested_file_name;
```

```
amt$sequence_number = 1 .. 9999
{ defaults to 0001 };

amt$skip_buffer_length = 1 .. amc$max_buffer_length;

amt$skip_count = 0 .. amc$file_byte_limit;

amt$skip_direction = (amc$forward, amc$backward);

amt$skip_option = (amc$skip_to_eor, amc$no_skip);

amt$skip_unit = (amc$skip_record, amc$skip_block,
  amc$skip_partition);

amt$sparse_key_control_effect = (amc$include_key_value,
  amc$exclude_key_value);

amt$statement_id_length = 1 ..
  amc$max_statement_id_length;

amt$statement_id_location = amt$page_width;

amt$statement_identifier = record
  length: amt$statement_id_length,
  location: amt$statement_id_location,
recend;

amt$store_attributes = array [1 .. * ] of
  amt$store_item;

amt$store_item = record
  case key: amt$file_attribute_keys of
  = amc$error_exit_procedure =
    error_exit_procedure: amt$error_exit_procedure,
  = amc$error_options =
    error_options: amt$error_options,
  = amc$label_exit_procedure =
    label_exit_procedure: amt$label_exit_procedure,
  = amc$label_options =
    label_options: amt$label_options,
  = amc$null_attribute =
    ,
  = amc$error_limit =
    error_limit: amt$error_limit,
  = amc$message_control =
    message_control: amt$message_control,
  casend,
recend;
```

```
amt$tape_mark_count = 1 .. amc$max_tape_mark_count;

amt$term_option = (amc$start, amc$continue,
  amc$terminate);

amt$transfer_count = amt$working_storage_length;

amt$unpack_block_header = record
  header_type: amt$block_header_type,
  block_length_as_read: amt$max_block_length,
  block_length_as_written: amt$max_block_length,
  block_number: amt$block_number,
  unused_bit_count: amt$unused_bit_count,
  block_status: amt$block_status,
recend;

amt$unused_bit_count = 0 .. 7;

amt$user_info = string (amc$max_user_info);

amt$version_number = 1 .. 99
{ defaults to 01 };

amt$vertical_print_density = 6 ..
  amc$max_lines_per_inch;

amt$volume_number = 1 .. amc$max_vol_number;

amt$volume_position = (amc$bov,
  amc$mid_bov_label_group, amc$after_tapemark,
  amc$mid_hdr_label_group, amc$mid_eof_label_group,
  amc$mid_eov_label_group, amc$eov);

amt$working_storage_length = ost$segment_length;
```

OS

Constants

```
osc$max_condition = 999999;
osc$max_name_size = 31;
osc$max_page_size = 65536;
osc$max_ring = 15, { Highest ring number (least }
{ privileged). };
osc$max_segment_length = osc$maximum_offset + 1;
osc$max_string_size = 256;
osc$maximum_offset = 7fffffff(16);
osc$maximum_segment = 0fff(16),

osc$min_ring = 1 { Lowest ring number (most }
{ privileged). };
osc$min_page_size = 512;

osc$null_name = '                                ';
osc$status_parameter_delimiter = CHR (31) { Unit }
{ Separator } ;
```

Ordinals

```
osc$invalid_ring = 0;
osc$os_ring_1 = 1 { Reserved for Operating System. };
osc$tmtr_ring = 2 { Task Monitor. };
osc$tsrv_ring = 3 { Task services. };
osc$sj_ring_1 = 4 { Reserved for system job. };
osc$sj_ring_2 = 5;
osc$sj_ring_3 = 6;
osc$application_ring_1 = 7 { Reserved for }
{ application subsystems. };
osc$application_ring_2 = 8;
osc$application_ring_3 = 9;
osc$application_ring_4 = 10;
osc$user_ring = 11 { Standard user task. };
osc$user_ring_1 = 12 { Reserved for user...O.S. }
{ requests available. };
osc$user_ring_2 = 13;
osc$user_ring_3 = 14 { Reserved for user...O.S. }
{ requests not available. };
osc$user_ring_4 = 15;
```

Types

```
ost$binary_unique_name = packed record
  processor: pmt$processor,
  year: 1980 .. 2047,
  month: 1 .. 12,
  day: 1 .. 31,
  hour: 0 .. 23,
  minute: 0 .. 59,
  second: 0 .. 59,
  sequence_number: 0 .. 9999999,
recend;

ost$clear_file_space = boolean;

ost$date_time = record
  year: 0 .. 255,
  month: 1 .. 12,
  day: 1 .. 31,
  hour: 0 .. 23,
  minute: 0 .. 59,
  second: 0 .. 59,
  millisecond: 0 .. 999,
recend;

ost$family_name = ost$name;

ost$key_lock = packed record
  global: boolean, { True if value is global key. }
  local: boolean, { True if value is local key. }
  value: ost$key_lock_value, { Key or lock value. }
recend;

ost$key_lock_value = 0 .. 3f(16);

ost$name = string (osc$max_name_size);

ost$name_size = 1 .. osc$max_name_size;

ost$page_size = osc$min_page_size ..
  osc$max_page_size;

ost$pva = packed record
  ring: ost$ring,
  seg: ost$segment,
  offset: ost$segment_offset,
recend;
```

```
ost$relative_pointer = - 7fffffff(16) ..
  7fffffff(16);

ost$ring = osc$invalid_ring ..
  osc$max_ring { Ring number };

ost$segment = 0 ..
  osc$maximum_segment { Segment number };

ost$segment_length = 0 .. osc$max_segment_length;

ost$segment_offset = - (osc$maximum_offset + 1) ..
  osc$maximum_offset;

ost$status = record
  case normal: boolean of
  = FALSE =
    identifier: string (2),
    condition: ost$status_condition,
    text: ost$string,
  casend,
recend;

ost$status_condition = 0 .. osc$max_condition;

ost$string = record
  size: ost$string_size,
  value: string (osc$max_string_size),
recend;

ost$string_index = 1 .. osc$max_string_size + 1;

ost$string_size = 0 .. osc$max_string_size;
```

```
ost$unique_name = record
  case boolean of
  = TRUE =
    value: ost$name,
  = FALSE =
    dollar_sign: string (1),
    sequence_number: string (7),
    p: string (1),
    processor_model_number: string (1),
    s: string (1),
    processor_serial_number: string (4),
    d: string (1),
    year: string (4),
    month: string (2),
    day: string (2),
    t: string (1),
    hour: string (2),
    minute: string (2),

    second: string (2),
  casend,
recend;

ost$user_identification = record
  user: ost$user_name,
  family: ost$family_name,
recend;

ost$user_name = ost$name;

ost$valid_ring = osc$min_ring ..
  osc$max_ring { valid ring Number };

ost$wait = (osc$wait, osc$nowait);

ost$wait_for_lock = (osc$wait_for_lock, osc$nowait_for_lock);
```

PF

Types

```
pft$application_info = string (osc$max_name_size);

pft$permit_options = (pfc$read, pfc$shorten,
  pfc$append, pfc$modify, pfc$execute, pfc$cycle,
  pfc$control);

pft$share_options = pfc$read .. pfc$execute;

pft$share_selections = set of pft$share_options;

pft$share_requirements = set of pft$share_options;

pft$usage_options = pfc$read .. pfc$execute;

pft$usage_selections = set of pft$usage_options;
```


PM

Types

```
pmt$cpu_model_number = (pmc$cpu_model_p1,
  pmc$cpu_model_p2, pmc$cpu_model_p3,
  pmc$cpu_model_p4);

pmt$cpu_serial_number = 0 .. 0ffff(16);

pmt$processor = record
  serial_number: pmt$cpu_serial_number,
  model_number: pmt$cpu_model_number,
recend;

pmt$processor_attributes = record
  model_number: pmt$cpu_model_number,
  serial_number: pmt$cpu_serial_number,
  page_size: ost$page_size,
recend,

pmt$program_name = ost$name;
```

RM

Constants

```
rmc$external_vsn_size = 6;
rmc$recorded_vsn_size = 6;
```

Types

```
rmt$external_vsn = string (rmc$external_vsn_size);

rmt$recorded_vsn = string (rmc$recorded_vsn_size);

rmt$volume_descriptor = record
  recorded_vsn: rmt$recorded_vsn,
  external_vsn: rmt$external_vsn,
recend;

rmt$volume_list = array [ * ] of
  rmt$volume_descriptor;
```

SM

Types

```
{ SMT$OWNCODE_PROCEDURE_TYPES - Pointer to owncode
{     procedure types for SORT and MERGE

own1to4_type= ^PROCEDURE (
               VAR return_code: integer;
               VAR reca: string(*);
               VAR rla: integer);

{ Collating sequence pointer.

smt$collating_sequence_pointer = ^string(256);

{ SMT$INFO_ARRAY - Description of static array
{      for SORT and MERGE

smt$info_array = array[1..16] OF integer;
```

Collation Tables D

This appendix describes how to use a collation table to specify how a key is ordered.

The collation table can be one of the NOS/VE predefined collation tables (listed at the end of this appendix) or a user-defined collation table.

The key to be ordered can be one of the following:

- The primary key of a keyed file. You specify the collation-table name as the value of the COLLATE_TABLE_NAME file attribute when creating the file (as described in chapter I-2).
- An alternate key of a keyed file. You specify the collation-table name as the value of the COLLATE_TABLE_NAME attribute of the alternate-key definition (as described in chapter I-2).
- A sort key. You can associate a key type name with the collation table using the Sort/Merge procedure calls SMP$DEFINE_USER_COLLATING_TABLE or SMP$LOAD_COLLATING_TABLE described in part II. The key type can then be used in a key field definition on an SMP$KEY call.

Using NOS/VE Predefined Collation Tables

To use one of the NOS/VE predefined collation tables listed at the end of this appendix, you specify the name of the predefined collation table as the collation-table name. Unlike user-defined collation table modules, use of NOS/VE predefined collation tables does not require the addition of an object library to the program-library list.

Sort/Merge Example:

To use the predefined collation table OSV$EBCDIC to define the key type MY_KEY_TYPE, you would include this call in the sequence of Sort/Merge procedure calls:

```
smp$load_collating_table('my_key_type', 'osv$ebcdic', status);
```

Then, to define the first 10 bytes of the record as a key field to be sorted in ascending order using the key type, you would include this Sort/Merge call:

```
smp$key(1, 10, 'my_key_type', 'a', status);
```

Keyed-File Example:

To use the predefined collation table OSV$EBCDIC to order the primary key of a new keyed file, you specify the key type as collated and the collate-table name as OSV$EBCDIC. This is done by initializing two attribute records in the attribute array for an AMP$FILE call before the new keyed file is opened or for the AMP$OPEN call that first opens the new keyed file.

```
[amc$key_type, amc$collated_key],
[amc$collate_table_name, 'OSV$EBCDIC'],
```

Using User-Defined Collation Tables

You can use any collation table stored in an object-library file if you have permission to read the file. To use the collation table, you perform these steps:

1. Specify the collation-table name in the program. (The name must be in the entry-point list of the object library as displayed by a DISPLAY_OBJECT_LIBRARY command.)
2. Add the object library to your program-library list using a SET_PROGRAM_ATTRIBUTE command before executing the program:

```
set_program_attribute add_library=$user.object_library
```

The process of storing a collation table in an object library is described in the Creating a Collation Table section.

For the purposes of these examples, assume another user has given you permission to read an object library file named .WIZARD.OBJECT_LIBRARY containing a collation table. The entry point for the collation table is named CASE_INSENSITIVE.

Sort/Merge Example:

To use the CASE_INSENSITIVE collation table:

1. In the Sort/Merge call sequence, specify the collation table as a key type and use the key type in a key-field definition:

```
smp$load_collating_table('my_key_type', 'case_insensitive',
                         status);
smp$key(1, 24, 'my_key_type', 'd', status);
```

2. Add the object library to your program-library list before executing the CYBIL program:

```
/set_program_attribute add_library=.wizard.object_library
```

Keyed-File Example:

To use the CASE_INSENSITIVE collation table to order a new alternate key of a keyed file:

1. Specify the key type as collated and the collate-table name as OSV$EBCDIC by initializing two attribute records in the optional_attributes array for the AMP$CREATE_KEY_DEFINITION call that defines the new alternate key.

```
[amc$key_type, amc$collated_key],
[amc$collate_table_name, 'OSV$EBCDIC'],
```

2. Add the object library to your program-library list before executing the CYBIL program:

```
/set_program_attribute add_library=.wizard.object_library
```

Creating a Collation Table

Besides using collation tables created by others, you can also create your own collation tables. The process of using your collation tables was described previously under Using User-Defined Collation Tables.

Creating your own collation table involves these steps:

1. Writing a source code module to initialize the collation table.
2. Compiling the source code module to create the object module.
3. Storing the object module in an object library.

Writing a Module to Initialize a Collation Table

A module to initialize a collation table must perform these steps:

1. Declare a 256-integer array.
2. Store an integer in each element of the array. The integer must be in the range 0 through 255.

The values stored in the array are the collating weights. The collating weight in an array element is the collating weight assigned to the ASCII character corresponding to that element.

How a Collation Table Works

To determine the correct values with which to initialize the collation table, you must understand how a collation table works.

As shown in figure D-1, each element in the collation table corresponds to an 8-bit character code. The first 128 elements correspond to the 128 characters in the ASCII character set (as listed in appendix B). For example, the element 0 in the table corresponds to the NUL character (character code 00 decimal). Element 65 corresponds to the A character (character code 65 decimal).

Figure D-2 shows how a collation table is initialized for the default ASCII collating sequence. As you can see, the element rank matches the element contents. For example, the element for character NUL (character code 00) contains 0. The element for character A (character code 65) contains 65.

Now, suppose we change two values in the initialized collation table in figure D-2. We change the A element to contain 66 (B) and the B element to contain 65 (A). This collating sequence would order all B characters as A characters and all A characters as B characters. A sort using the collating sequence would sort all B characters before all A characters.

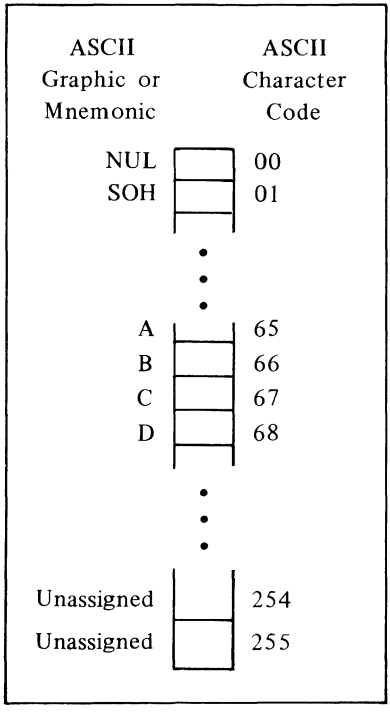


Figure D-1. Uninitialized Collation Table

Or, suppose we change the initialized collation table so that the A element contains 65 (A) and the B element also contains 65 (A). This collating sequence would order all A characters as A characters and all B characters as A characters. A sort using the collating sequence would intermix A and B characters.

NOTE

Be careful when choosing the collating sequence to order the primary key of a keyed file. A collating sequence that assigns equal values to different characters reduces the possible unique key values.

If the key values are collated equally, the values are no longer unique in the file. For example, if B is collated as A, the key value B is a duplicate of key value A.

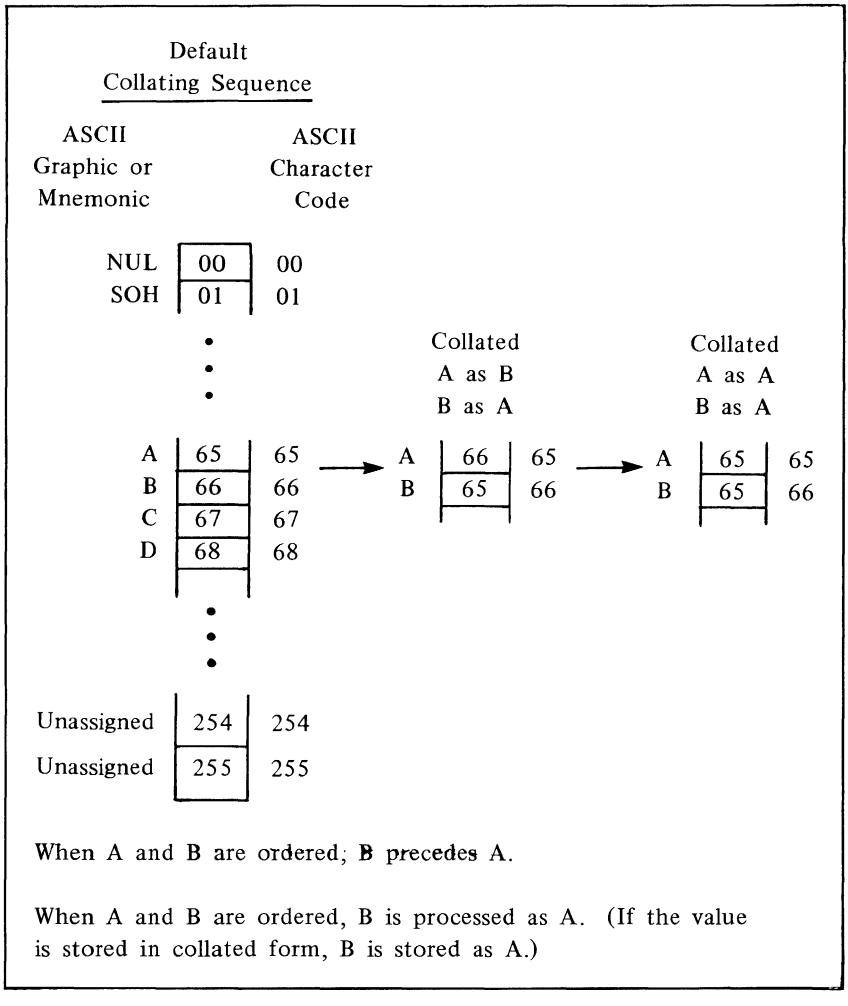


Figure D-2. Collation Table Initialized to the Default ASCII Collating Sequence

CYBIL Collation Table Initialization Examples

A CYBIL module to initialize a collation table declares a 256-element array variable and assigns a value to each element.

NOTE

The array variable must be assigned the XDCL attribute so that the name is an entry point to the module. A module can define more than one collation table by declaring and initializing more than one XDCL array variable.

Figure D-3 shows a CYBIL module named MY_MODULE that initializes an XDCL variable named CASE_INSENSITIVE. It assigns the collating weight for the space character (32) to all elements except the elements corresponding to letters. Each lowercase letter is to be ordered the same as the corresponding uppercase letter (a the same as A, b the same as B, and so forth).

```
MODULE my_module;

VAR
case_insensitive: [STATIC,READ,XDCL] ARRAY [CHAR] OF 0..255 :=

[ { Collating weights for the first 65 non-letter characters }
  REP 65 OF 32,

  { Collating weights for the uppercase letters }
    {A} 65, {B} 66, {C} 67, {D} 68, {E} 69, {F} 70, {G} 71,
    {H} 72, {I} 73, {J} 74, {K} 75, {L} 76, {M} 77, {N} 78,
    {O} 79, {P} 80, {Q} 81, {R} 82, {S} 83, {T} 84, {U} 85,
    {V} 86, {W} 87, {X} 88, {Y} 89, {Z} 90,

  { Collating weights for the next 6 non-letter characters }
    REP 6 OF 32,

  { Collating weights for the lowercase letters }
    {a} 65, {b} 66, {c} 67, {d} 68, {e} 69, {f} 70, {g} 71,
    {h} 72, {i} 73, {j} 74, {k} 75, {l} 76, {m} 77, {n} 78,
    {o} 79, {p} 80, {q} 81, {r} 82, {s} 83, {t} 84, {u} 85,
    {v} 86, {w} 87, {x} 88, {y} 89, {z} 90,

  { Collating weights for the last 133 non-letter characters }
    REP 133 OF 32 ];

MODEND;
```

Figure D-3. CASE_INSENSITIVE Collating Sequence Initialization Module

Sort/Merge Example:

If Sort/Merge used the collation table from figure D-3, it would sort characters as follows:

```
Unordered: 10]garbageGARBAGEgarbage9815];]
Ordered:   10]9815];]aaAAaabBbeEeggGGggrRr
```

Keyed-File Example:

If a keyed file used the collation table from figure D-3, all nonalphabetic key values would be duplicates. Uppercase and lowercase letters would be collated the same, so the key value ABCD would be a duplicate of the key value abcd.

Storing a Module in an Object Library

Source module compilation writes an object module on an object file. You then use the SCL command utility CREATE_OBJECT_LIBRARY to create an object library containing the module. (The CREATE_OBJECT_LIBRARY utility is described in detail in the SCL Object Code Management manual.)

For this example, assume that you have written a CYBIL module (such as the one in figure D-3) to initialize a collation table and that your source text is in file $USER.SOURCE. The following commands compile the program and then store the module on file $USER.COLLATION_LIBRARY

```
/cybil input=$user.source binary_object=object_file ..
../list=list_file
/create_object_library
COL/add_module library=object_file
COL/generate_library library=$user.collation_library
COL/quit
/
```

NOS/VE Predefined Collation Table Listings

The collating sequences of the predefined collation tables are listed in tables D-1 through D-11.

Several of the predefined collation tables have two variants, FOLDED and STRICT. The variants FOLDED and STRICT indicate two different mappings of the characters not in the 63 or 64 characters of the original CYBER 170 collating sequence.

- A strict mapping maps all characters not in the original 64- or 63-character set to the space character.
- A folded mapping maps some of these characters to the space character, but not others. (For the exact mapping, see the collating sequence in the table.)

The predefined collation tables are for these collating sequences:

Collating Sequence	Predefined Collation Table
CYBER 170 FTN5 default	OSV$ASCII6_FOLDED and OSV$ASCII6_STRICT
CYBER 170 COBOL5 default	OSV$COBOL6_FOLDED and OSV$COBOL6_STRICT
CYBER 170 63-character display code	OSV$DISPLAY63_FOLDED and OSV$DISPLAY63_STRICT
CYBER 170 64-character display code	OSV$DISPLAY64_FOLDED and OSV$DISPLAY64_STRICT
Full EBCDIC	OSV$EBCDIC
EBCDIC 6-bit subset supported by CYBER 170 COBOL5 and SORT5	OSV$EBCDIC_FOLDED and OSV$EBCDIC_STRICT

Sort/Merge uses predefined collation tables for its predefined collating sequences as follows:

Key Type	Predefined Collation Table
ASCII6	OSV$ASCII6_FOLDED
COBOL6	OSV$COBOL6_FOLDED
DISPLAY	OSV$DISPLAY64_FOLDED
EBCDIC	OSV$EBCDIC
EBCDIC6	OSV$EBCDIC6_FOLDED

The Sort/Merge key type ASCII uses the default ASCII collating sequence; it does not use any of the predefined collating sequences listed in this appendix.

Table D-1. OSV$ASCII6_FOLDED Collating Sequence

The ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40,60	@,`	Commercial at, grave accent
33	41,61	A,a	Uppercase A, lowercase a
34	42,62	B,b	Uppercase B, lowercase b
35	43,63	C,c	Uppercase C, lowercase c
36	44,64	D,d	Uppercase D, lowercase d
37	45,65	E,e	Uppercase E, lowercase e
38	46,66	F,f	Uppercase F, lowercase f
39	47,67	G,g	Uppercase G, lowercase g

(Continued)

Table D-1. OSV$ASCII6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48,68	H,h	Uppercase H, lowercase h
41	49,69	I,i	Uppercase I, lowercase i
42	4A,6A	J,j	Uppercase J, lowercase j
43	4B,6B	K,k	Uppercase K, lowercase k
44	4C,6C	L,l	Uppercase L, lowercase l
45	4D,6D	M,m	Uppercase M, lowercase m
46	4E,6E	N,n	Uppercase N, lowercase n
47	4F,6F	O,o	Uppercase O, lowercase o
48	50,70	P,p	Uppercase P, lowercase p
49	51,71	Q,q	Uppercase Q, lowercase q
50	52,72	R,r	Uppercase R, lowercase r
51	53,73	S,s	Uppercase S, lowercase s
52	54,74	T,t	Uppercase T, lowercase t
53	55,75	U,u	Uppercase U, lowercase u
54	56,76	V,v	Uppercase V, lowercase v
55	57,77	W,w	Uppercase W, lowercase w
56	58,78	X,x	Uppercase X, lowercase x
57	59,79	Y,y	Uppercase Y, lowercase y
58	5A,7A	Z,z	Uppercase Z, lowercase z
59	5B,7B	[,{	Opening bracket, opening brace
60	5C,7C	\ , |	Reverse slant, vertical line
61	5D,7D	] , }	Closing bracket, closing brace
62	5E,7E	^ , ~	Circumflex, tilde
63	5F	_	Underline

Table D-2. OSV$ASCII6_STRICT Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40	@	Commercial at
33	41	A	Uppercase A
34	42	B	Uppercase B
35	43	C	Uppercase C
36	44	D	Uppercase D
37	45	E	Uppercase E
38	46	F	Uppercase F
39	47	G	Uppercase G

(Continued)

Table D-2. OSV$ASCII6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48	H	Uppercase H
41	49	I	Uppercase I
42	4A	J	Uppercase J
43	4B	K	Uppercase K
44	4C	L	Uppercase L
45	4D	M	Uppercase M
46	4E	N	Uppercase N
47	4F	O	Uppercase O
48	50	P	Uppercase P
49	51	Q	Uppercase Q
50	52	R	Uppercase R
51	53	S	Uppercase S
52	54	T	Uppercase T
53	55	U	Uppercase U
54	56	V	Uppercase V
55	57	W	Uppercase W
56	58	X	Uppercase X
57	59	Y	Uppercase Y
58	5A	Z	Uppercase Z
59	5B	[	Opening bracket
60	5C	\	Reverse slant
61	5D	]	Closing bracket
62	5E	^	Circumflex
63	5F	_	Underline

Table D-3. OSV$COBOL6_FOLDED Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40,60	@,`	Commercial at, grave accent
02	25	%	Percent sign
03	5B,7B	[,{	Opening bracket, opening brace
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C,7C	\ , |	Reverse slant, vertical line
11	5E,7E	^ ,~	Circumflex, tilde
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41,61	A,a	Uppercase A, lowercase a
26	42,62	B,b	Uppercase B, lowercase b
27	43,63	C,c	Uppercase C, lowercase c
28	44,64	D,d	Uppercase D, lowercase d
29	45,65	E,e	Uppercase E, lowercase e
30	46,66	F,f	Uppercase F, lowercase f
31	47,67	G,g	Uppercase G, lowercase g
32	48,68	H,h	Uppercase H, lowercase h
33	49,69	I,i	Uppercase I, lowercase i
34	21	!	Exclamation point
35	4A,6A	J,j	Uppercase J, lowercase j
36	4B,6B	K,k	Uppercase K, lowercase k
37	4C,6C	L,l	Uppercase L, lowercase l
38	4D,6D	M,m	Uppercase M, lowercase m
39	4E,6E	N,n	Uppercase N, lowercase n

(Continued)

Table D-3. OSV$COBOL6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4F,6F	O,o	Uppercase O, lowercase o
41	50,70	P,p	Uppercase P, lowercase p
42	51,71	Q,q	Uppercase Q, lowercase q
43	52,72	R,r	Uppercase R, lowercase r
44	5D,7D	] , }	Closing bracket, closing brace
45	53,73	S,s	Uppercase S, lowercase s
46	54,74	T,t	Uppercase T, lowercase t
47	55,75	U,u	Uppercase U, lowercase u
48	56,76	V,v	Uppercase V, lowercase v
49	57,77	W,w	Uppercase W, lowercase w
50	58,78	X,x	Uppercase X, lowercase x
51	59,79	Y,y	Uppercase Y, lowercase y
52	5A,7A	Z,z	Uppercase Z, lowercase z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table D-4. OSV$COBOL6_STRICT Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40	@	Commercial at
02	25	%	Percent sign
03	5B	[	Opening bracket
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C	\	Reverse slant
11	5E	^	Circumflex
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41	A	Uppercase A
26	42	B	Uppercase B
27	43	C	Uppercase C
28	44	D	Uppercase D
29	45	E	Uppercase E
30	46	F	Uppercase F
31	47	G	Uppercase G
32	48	H	Uppercase H
33	49	I	Uppercase I
34	21	!	Exclamation point
35	4A	J	Uppercase J
36	4B	K	Uppercase K
37	4C	L	Uppercase L
38	4D	M	Uppercase M
39	4E	N	Uppercase N

(Continued)

Table D-4. OSV$COBOL6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4F	O	Uppercase O
41	50	P	Uppercase P
42	51	Q	Uppercase Q
43	52	R	Uppercase R
44	5D	]	Closing bracket
45	53	S	Uppercase S
46	54	T	Uppercase T
47	55	U	Uppercase U
48	56	V	Uppercase V
49	57	W	Uppercase W
50	58	X	Uppercase X
51	59	Y	Uppercase Y
52	5A	Z	Uppercase Z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table D-5. OSV$DISPLAY63_FOLDED Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	41,61	A,a	Uppercase A, lowercase a
01	42,62	B,b	Uppercase B, lowercase b
02	43,63	C,c	Uppercase C, lowercase c
03	44,64	D,d	Uppercase D, lowercase d
04	45,65	E,e	Uppercase E, lowercase e
05	46,66	F,f	Uppercase F, lowercase f
06	47,67	G,g	Uppercase G, lowercase g
07	48,68	H,h	Uppercase H, lowercase h
08	49,69	I,i	Uppercase I, lowercase i
09	4A,6A	J,j	Uppercase J, lowercase j
10	4B,6B	K,k	Uppercase K, lowercase k
11	4C,6C	L,l	Uppercase L, lowercase l
12	4D,6D	M,m	Uppercase M, lowercase m
13	4E,6E	N,n	Uppercase N, lowercase n
14	4F,6F	O,o	Uppercase O, lowercase o
15	50,70	P,p	Uppercase P, lowercase p
16	51,71	Q,q	Uppercase Q, lowercase q
17	52,72	R,r	Uppercase R, lowercase r
18	53,73	S,s	Uppercase S, lowercase s
19	54,74	T,t	Uppercase T, lowercase t
20	55,75	U,u	Uppercase U, lowercase u
21	56,76	V,v	Uppercase V, lowercase v
22	57,77	W,w	Uppercase W, lowercase w
23	58,78	X,x	Uppercase X, lowercase x
24	59,79	Y,y	Uppercase Y, lowercase y
25	5A,7A	Z,z	Uppercase Z, lowercase z
26	30	0	Zero
27	31	1	One
28	32	2	Two
29	33	3	Three
30	34	4	Four
31	35	5	Five
32	36	6	Six
33	37	7	Seven
34	38	8	Eight
35	39	9	Nine
36	2B	+	Plus
37	2D	-	Hyphen
38	2A	*	Asterisk
39	2F	/	Slant

(Continued)

Table D-5. OSV$DISPLAY63_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	24	$	Dollar sign
43	3D	=	Equals
44	20	SP	Space
45	2C	,	Comma
46	2E	.	Period
47	23	#	Number sign
48	5B,7B	[, {	Opening bracket, opening brace
49	5D,7D	] , }	Closing bracket, closing brace
50	3A	:	Colon
51	22	"	Quotation marks
52	5F	_	Underline
53	21	!	Exclamation point
54	26	&	Ampersand
55	27	'	Apostrophe
56	3F	?	Question mark
57	3C	<	Less than
58	3E	>	Greater than
59	40,60	@,`	Commercial at, grave accent
60	5C,7C	\ , |	Reverse slant, vertical line
61	5E,7E	^ ,~	Circumflex, tilde
62	3B	;	Semicolon

Table D-6. OSV$DISPLAY63_STRICT Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	41	A	Uppercase A
01	42	B	Uppercase B
02	43	C	Uppercase C
03	44	D	Uppercase D
04	45	E	Uppercase E
05	46	F	Uppercase F
06	47	G	Uppercase G
07	48	H	Uppercase H
08	49	I	Uppercase I
09	4A	J	Uppercase J
10	4B	K	Uppercase K
11	4C	L	Uppercase L
12	4D	M	Uppercase M
13	4E	N	Uppercase N
14	4F	O	Uppercase O
15	50	P	Uppercase P
16	51	Q	Uppercase Q
17	52	R	Uppercase R
18	53	S	Uppercase S
19	54	T	Uppercase T
20	55	U	Uppercase U
21	56	V	Uppercase V
22	57	W	Uppercase W
23	58	X	Uppercase X
24	59	Y	Uppercase Y
25	5A	Z	Uppercase Z
26	30	0	Zero
27	31	1	One
28	32	2	Two
29	33	3	Three
30	34	4	Four
31	35	5	Five
32	36	6	Six
33	37	7	Seven
34	38	8	Eight
35	39	9	Nine
36	2B	+	Plus
37	2D	-	Hyphen
38	2A	*	Asterisk
39	2F	/	Slant

(Continued)

Table D-6. OSV$DISPLAY63_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	24	$	Dollar sign
43	3D	=	Equals
44	20	SP	Space
45	2C	,	Comma
46	2E	.	Period
47	23	#	Number sign
48	5B	[	Opening bracket
49	5D	]	Closing bracket
50	3A	:	Colon
51	22	"	Quotation marks
52	5F	_	Underline
53	21	!	Exclamation point
54	26	&	Ampersand
55	27	'	Apostrophe
56	3F	?	Question mark
57	3C	<	Less than
58	3E	>	Greater than
59	40	@	Commercial at
60	5C	\	Reverse slant
61	5E	^	Circumflex
62	3B	;	Semicolon

Table D-7. OSV$DISPLAY64_FOLDED Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41,61	A,a	Uppercase A, lowercase a
02	42,62	B,b	Uppercase B, lowercase b
03	43,63	C,c	Uppercase C, lowercase c
04	44,64	D,d	Uppercase D, lowercase d
05	45,65	E,e	Uppercase E, lowercase e
06	46,66	F,f	Uppercase F, lowercase f
07	47,67	G,g	Uppercase G, lowercase g
08	48,68	H,h	Uppercase H, lowercase h
09	49,69	I,i	Uppercase I, lowercase i
10	4A,6A	J,j	Uppercase J, lowercase j
11	4B,6B	K,k	Uppercase K, lowercase k
12	4C,6C	L,l	Uppercase L, lowercase l
13	4D,6D	M,m	Uppercase M, lowercase m
14	4E,6E	N,n	Uppercase N, lowercase n
15	4F,6F	O,o	Uppercase O, lowercase o
16	50,70	P,p	Uppercase P, lowercase p
17	51,71	Q,q	Uppercase Q, lowercase q
18	52,72	R,r	Uppercase R, lowercase r
19	53,73	S,s	Uppercase S, lowercase s
20	54,74	T,t	Uppercase T, lowercase t
21	55,75	U,u	Uppercase U, lowercase u
22	56,76	V,v	Uppercase V, lowercase v
23	57,77	W,w	Uppercase W, lowercase w
24	58,78	X,x	Uppercase X, lowercase x
25	59,79	Y,y	Uppercase Y, lowercase y
26	5A,7A	Z,z	Uppercase Z, lowercase z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk

(Continued)

Table D-7. OSV$DISPLAY64_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B,7B	[, {	Opening bracket, opening brace
50	5D,7D	] , }	Closing bracket, closing brace
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@ , `	Commercial at, grave accent
61	5C,7C	\ , |	Reverse slant, vertical line
62	5E,7E	^ , ~	Circumflex, tilde
63	3B	;	Semicolon

Table D-8. OSV$DISPLAY64_STRICT Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41	A	Uppercase A
02	42	B	Uppercase B
03	43	C	Uppercase C
04	44	D	Uppercase D
05	45	E	Uppercase E
06	46	F	Uppercase F
07	47	G	Uppercase G
08	48	H	Uppercase H
09	49	I	Uppercase I
10	4A	J	Uppercase J
11	4B	K	Uppercase K
12	4C	L	Uppercase L
13	4D	M	Uppercase M
14	4E	N	Uppercase N
15	4F	O	Uppercase O
16	50	P	Uppercase P
17	51	Q	Uppercase Q
18	52	R	Uppercase R
19	53	S	Uppercase S
20	54	T	Uppercase T
21	55	U	Uppercase U
22	56	V	Uppercase V
23	57	W	Uppercase W
24	58	X	Uppercase X
25	59	Y	Uppercase Y
26	5A	Z	Uppercase Z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk

(Continued)

Table D-8. OSV$DISPLAY64_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Table D-9. OSV$EBCDIC Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
000	00	NUL	Null
001	01	SOH	Start of heading
002	02	STX	Start of text
003	03	ETX	End of text
004	9C	—	Unassigned
005	09	HT	Horizontal tabulation
006	86	—	Unassigned
007	7F	DEL	Delete
008	97	—	Unassigned
009	8D	—	Unassigned
010	8E	—	Unassigned
011	0B	VT	Vertical tabulation
012	0C	FF	Form feed
013	0D	CR	Carriage return
014	0E	SO	Shift out
015	0F	SI	Shift in
016	10	DLE	Data link escape
017	11	DC1	Device control 1
018	12	DC2	Device control 2
019	13	DC3	Device control 3
020	9D	—	Unassigned
021	85	—	Unassigned
022	08	BS	Backspace
023	87	—	Unassigned
024	18	CAN	Cancel
025	19	EM	End of medium
026	92	—	Unassigned
027	8F	—	Unassigned
028	1C	FS	File separator
029	1D	GS	Group separator
030	1E	RS	Record separator
031	1F	US	Unit separator
032	80	—	Unassigned
033	81	—	Unassigned
034	82	—	Unassigned
035	83	—	Unassigned
036	84	—	Unassigned
037	0A	LF	Line feed
038	17	ETB	End of transmission block
039	1B	ESC	Escape
040	88	—	Unassigned
041	89	—	Unassigned
042	8A	—	Unassigned
043	8B	—	Unassigned

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
044	8C	—	Unassigned
045	05	ENQ	Enquiry
046	06	ACK	Acknowledge
047	07	BEL	Bell
048	90	—	Unassigned
049	91	—	Unassigned
050	16	SYN	Synchronous idle
051	93	—	Unassigned
052	94	—	Unassigned
053	95	—	Unassigned
054	96	—	Unassigned
055	04	EOT	End of transmission
056	98	—	Unassigned
057	99	—	Unassigned
058	9A	—	Unassigned
059	9B	—	Unassigned
060	14	DC4	Device control 4
061	15	NAK	Negative acknowledge
062	9E	—	Unassigned
063	1A	SUB	Substitute
064	20	SP	Space
065	A0	—	Unassigned
066	A1	—	Unassigned
067	A2	—	Unassigned
068	A3	—	Unassigned
069	A4	—	Unassigned
070	A5	—	Unassigned
071	A6	—	Unassigned
072	A7	—	Unassigned
073	A8	—	Unassigned
074	5B	[	Opening bracket
075	2E	.	Period
076	3C	<	Less than
077	28	(	Opening parenthesis
078	2B	+	Plus
079	21	!	Exclamation point
080	26	&	Ampersand
081	A9	—	Unassigned
082	AA	—	Unassigned
083	AB	—	Unassigned
084	AC	—	Unassigned
085	AD	—	Unassigned
086	AE	—	Unassigned
087	AF	—	Unassigned

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
088	B0	—	Unassigned
089	B1	—	Unassigned
090	5D	]	Closing bracket
091	24	$	Dollar sign
092	2A	*	Asterisk
093	29	)	Closing parenthesis
094	3B	;	Semicolon
095	5E	^	Circumflex
096	2D	-	Hyphen
097	2F	/	Slant
098	B2	—	Unassigned
099	B3	—	Unassigned
100	B4	—	Unassigned
104	B8	—	Unassigned
105	B9	—	Unassigned
106	7C	|	Vertical line
107	2C	,	Comma
108	25	%	Percent sign
109	5F	_	Underline
110	3E	>	Greater than
111	3F	?	Question mark
112	BA	—	Unassigned
113	BB	—	Unassigned
114	BC	—	Unassigned
115	BD	—	Unassigned
116	BE	—	Unassigned
117	BF	—	Unassigned
118	C0	—	Unassigned
119	C1	—	Unassigned
120	C2	—	Unassigned
121	60	`	Grave accent
122	3A	:	Colon
123	23	#	Number sign
124	40	@	Commercial at
125	27	'	Apostrophe
126	3D	=	Equals
127	22	"	Quotation marks
128	C3	—	Unassigned
129	61	a	Lowercase a
130	62	b	Lowercase b
131	63	c	Lowercase c

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
132	64	d	Lowercase d
133	65	e	Lowercase e
134	66	f	Lowercase f
135	67	g	Lowercase g
136	68	h	Lowercase h
137	69	i	Lowercase i
138	C4	—	Unassigned
139	C5	—	Unassigned
140	C6	—	Unassigned
141	C7	—	Unassigned
142	C8	—	Unassigned
143	C9	—	Unassigned
144	CA	—	Unassigned
145	6A	j	Lowercase j
146	6B	k	Lowercase k
147	6C	l	Lowercase l
148	6D	m	Lowercase m
149	6E	n	Lowercase n
150	6F	o	Lowercase o
151	70	p	Lowercase p
152	71	q	Lowercase q
153	72	r	Lowercase r
154	CB	—	Unassigned
155	CC	—	Unassigned
156	CD	—	Unassigned
157	CE	—	Unassigned
158	CF	—	Unassigned
159	D0	—	Unassigned
160	D1	—	Unassigned
161	7E	—	Unassigned
162	73	s	Lowercase s
163	74	t	Lowercase t
164	75	u	Lowercase u
165	76	v	Lowercase v
166	77	w	Lowercase w
167	78	x	Lowercase x
168	79	y	Lowercase y
169	7A	z	Lowercase z
170	D2	—	Unassigned
171	D3	—	Unassigned
172	D4	—	Unassigned
173	D5	—	Unassigned
174	D6	—	Unassigned
175	D7	—	Unassigned

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
176	D8	—	Unassigned
177	D9	—	Unassigned
178	DA	—	Unassigned
179	DB	—	Unassigned
180	DC	—	Unassigned
181	DD	—	Unassigned
182	DE	—	Unassigned
183	DF	—	Unassigned
184	E0	—	Unassigned
185	E1	—	Unassigned
186	E2	—	Unassigned
187	E3	—	Unassigned
188	E4	—	Unassigned
189	E5	—	Unassigned
190	E6	—	Unassigned
191	E7	—	Unassigned
192	7B	{	Opening brace
193	41	A	Uppercase A
194	42	B	Uppercase B
195	43	C	Uppercase C
196	44	D	Uppercase D
197	45	E	Uppercase E
198	46	F	Uppercase F
199	47	G	Uppercase G
200	48	H	Uppercase H
201	49	I	Uppercase I
202	E8	—	Unassigned
203	E9	—	Unassigned
204	EA	—	Unassigned
205	EB	—	Unassigned
206	EC	—	Unassigned
207	ED	—	Unassigned
208	7D	}	Closing brace
209	4A	J	Uppercase J
210	4B	K	Uppercase K
211	4C	L	Uppercase L
212	4D	M	Uppercase M
213	4E	N	Uppercase N
214	4F	O	Uppercase O
215	50	P	Uppercase P
216	51	Q	Uppercase Q
217	52	R	Uppercase R
218	EE	—	Unassigned
219	EF	—	Unassigned

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
220	F0	—	Unassigned
221	F1	—	Unassigned
222	F2	—	Unassigned
223	F3	—	Unassigned
224	5C	\	Reverse slant
225	9F	—	Unassigned
226	53	S	Uppercase S
227	54	T	Uppercase T
228	55	U	Uppercase U
229	56	V	Uppercase V
230	57	W	Uppercase W
231	58	X	Uppercase X
232	59	Y	Uppercase Y
233	5A	Z	Uppercase Z
234	F4	—	Unassigned
235	F5	—	Unassigned
236	F6	—	Unassigned
237	F7	—	Unassigned
238	F8	—	Unassigned
239	F9	—	Unassigned
240	30	0	Zero
241	31	1	One
242	32	2	Two
243	33	3	Three
244	34	4	Four
245	35	5	Five
246	36	6	Six
247	37	7	Seven
248	38	8	Eight
249	39	9	Nine
250	FA	—	Unassigned
251	FB	—	Unassigned
252	FC	—	Unassigned
253	FD	—	Unassigned
254	FE	—	Unassigned
255	FF	—	Unassigned

Table D-10. OSV$EBCDIC6_FOLDED Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E,7E	^ ,~	Circumflex, tilde
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40,60	@ ,`	Commercial at, grave accent
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B,7B	[, {	Opening bracket, opening brace
26	41,61	A,a	Uppercase A, lowercase a
27	42,62	B,b	Uppercase B, lowercase b
28	43,63	C,c	Uppercase C, lowercase c
29	44,64	D,d	Uppercase D, lowercase d
30	45,65	E,e	Uppercase E, lowercase e
31	46,66	F,f	Uppercase F, lowercase f
32	47,67	G,g	Uppercase G, lowercase g
33	48,68	H,h	Uppercase H, lowercase h
34	49,69	I,i	Uppercase I, lowercase i
35	5D,7D	] , }	Closing bracket, closing brace
36	4A,6A	J,j	Uppercase J, lowercase j
37	4B,6B	K,k	Uppercase K, lowercase k
38	4C,6C	L,l	Uppercase L, lowercase l
39	4D,6D	M,m	Uppercase M, lowercase m

(Continued)

Table D-10. OSV$EBCDIC6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4E,6E	N,n	Uppercase N, lowercase n
41	4F,6F	O,o	Uppercase O, lowercase o
42	50,70	P,p	Uppercase P, lowercase p
43	51,71	Q,q	Uppercase Q, lowercase q
44	52,72	R,r	Uppercase R, lowercase r
45	5C,7C	\, |	Reverse slant, vertical line
46	53,73	S,s	Uppercase S, lowercase s
47	54,74	T,t	Uppercase T, lowercase t
48	55,75	U,u	Uppercase U, lowercase u
49	56,76	V,v	Uppercase V, lowercase v
50	57,77	W,w	Uppercase W, lowercase w
51	58,78	X,x	Uppercase X, lowercase x
52	59,79	Y,y	Uppercase Y, lowercase y
53	5A,7A	Z,z	Uppercase Z, lowercase z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table D-11. OSV$EBCDIC6_STRICT Collating Sequence

The ASCII codes not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexcadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E	^	Circumflex
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40	@	Commercial at
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B	[	Opening bracket
26	41	A	Uppercase A
27	42	B	Uppercase B
28	43	C	Uppercase C
29	44	D	Uppercase D
30	45	E	Uppercase E
31	46	F	Uppercase F
32	47	G	Uppercase G
33	48	H	Uppercase H
34	49	I	Uppercase I
35	5D	]	Closing bracket
36	4A	J	Uppercase J
37	4B	K	Uppercase K
38	4C	L	Uppercase L
39	4D	M	Uppercase M

(Continued)

Table D-11. OSV$EBCDIC6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4E	N	Uppercase N
41	4F	O	Uppercase O
42	50	P	Uppercase P
43	51	Q	Uppercase Q
44	52	R	Uppercase R
45	5C	\	Reverse slant
46	53	S	Uppercase S
47	54	T	Uppercase T
48	55	U	Uppercase U
49	56	V	Uppercase V
50	57	W	Uppercase W
51	58	X	Uppercase X
52	59	Y	Uppercase Y
53	5A	Z	Uppercase Z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Common Procedures E

Each CYBIL procedure call specified a status variable in which the completion status of the call is returned. After the call, the program checks the status returned. The examples in part I call the p#inspect_status_ variable procedure to check the status after each call. Use of the p#inspect_status_variable procedure also requires calls to p#start_report_generation and p#stop_report_generation at the beginning and end of the program, respectively.

The program examples in part I copy a deck named COMPROC to include the common procedures in the program. The following is a listing of the text stored in deck COMPROC.

```
?? PUSH (LIST := OFF) ??
*copyc amp$close
*copyc amp$open
*copyc amp$put_next
*copyc osp$format_message
?? POP ??

  CONST
    line_length = 137 ; { 1 carriage control characters + 136 data
                        { characters }

  SECTION s#storage_area        : READ ; { read-only memory  }
  SECTION s#global_holding_area : WRITE  { read-write memory }
  VAR
    error_count            : [STATIC, s#global_holding_area]
                               INTEGER := -1 , { global error counter }
    report_file_name       : [STATIC, READ, s#storage_area]
                               AMT$LOCAL_FILE_NAME := '$output' ,
    report_file_identifier : [STATIC, s#global_holding_area]
                               AMT$FILE_IDENTIFIER ,
    text_index             : [STATIC, s#global_holding_area]
                               1 .. line_length+1 , { line buffer pointer }
    text_line              : [STATIC, s#global_holding_area]
                               STRING ( line_length ) , { line buffer }
    status                 : [STATIC, s#global_holding_area]
                               OST$STATUS ; { global status variable }
```

```
{ ------------------------------------------------------------------------ }
{ This routine, P#START_REPORT_GENERATION, takes care of initialization }
{ details.  It sets the error tally to zero and prepares the report file to }
{ receive messages issued by the other procedures. }
{ ------------------------------------------------------------------------ }

  PROCEDURE p#start_report_generation (startup_message : STRING ( * )) ;

    VAR
      file_access_selection_p : ^ ARRAY [1 .. *] OF AMT$ACCESS_SELECTION ;
                                                  { used by AMP$OPEN_FILE }

    error_count := -0 ;                      { initialize error counting }

    ALLOCATE file_access_selection_p : [1 .. 1] ;
    file_access_selection_p^[01].KEY           := AMC$OPEN_POSITION ;
    file_access_selection_p^[01].OPEN_POSITION := AMC$OPEN_NO_POSITIONING ;
                                     { must be positioned for append access }
    AMP$OPEN (report_file_name, AMC$RECORD, file_access_selection_p,
              report_file_identifier, status) ;
    FREE file_access_selection_p ;

    text_index := 1 ;
    text_line(text_index, 1) := '0' ;            { carriage control character }
    text_index := text_index + 1 ;
    p#put_m (TRUE, startup_message) ;

  PROCEND p#start_report_generation ;


{ ------------------------------------------------------------------------ }
{ Routine P#STOP_REPORT_GENERATION does wrap-up activity.  The error tally  }
{ is printed out at this point.                                             }
{ ------------------------------------------------------------------------ }

  PROCEDURE p#stop_report_generation (shutdown_message : STRING ( * )) ;

    VAR
      pencil : INTEGER ,                         { formatting area length }
      paper  : STRING ( 75 ) ;                   { formatting area        }

    IF error_count = 0
      THEN
      p#put_m (TRUE, 'No error has been found by the program.') ;
    ELSE
      STRINGREP (paper, pencil, 'This program has discovered ',
                 error_count, ' error situation(s).') ;
      p#put_m (TRUE, paper(1, pencil)) ;
    IFEND ;

    p#put_m (TRUE, shutdown_message) ;

    AMP$CLOSE (report_file_identifier, status) ;

  PROCEND p#stop_report_generation ;
```

```
{ ------------------------------------------------------------------------ }
{ P#PUT_M places the parameter message_string onto the reporting file, taking }
{ care to wrap around any long message text by splitting it onto additional   }
{ physical lines.  Data is appended at the current character position of      }
{ text_line; it doesn't automatically start in column 1.  The parameter      }
{ new_line_flag tells whether or not end-of-line should follow the message.  }
{ Any unprintable character is translated into '?'.                          }
{ ------------------------------------------------------------------------ }

  PROCEDURE p#put_m (new_line_flag : boolean ;
                     message_string : string ( * <= 500 )) ;

    VAR
      garbage_eliminator_table : [s#storage_area, STATIC, READ]
        string ( 256 ) := '????????????????????????????????????'
        CAT ' !"#$%&''()*+,-./0123456789:;<=>?@'
        CAT 'ABCDEFGHIJKLMNOPQRSTUVWXYZ[ ]^_`'
        CAT 'abcdefghijklmnopqrstuvwxyz{ }~?'
        CAT '????????????????????????????????????????????????????????????????'
        CAT '????????????????????????????????????????????????????????????????',
      string_position_locator  : 1 .. line_length ,

     { Dummy variables, not used }
      file_byte_address_x      : amt$file_byte_address ,
      status_x                 : ost$status ;

    IF error_count = -1                      { initialization forgotten ? }
      THEN
      p#start_report_generation ('Program ?? is starting.') ;
      error_count := error_count + 1 ;
      p#put_m (TRUE,
               'Error Detected!  P#PUT_M was invoked without being preceded
                by P#START_REPORT_GENERATION.') ;
    ELSE
      IF (text_index + STRLENGTH(message_string) - 1) = line_length
        THEN
        #TRANSLATE (garbage_eliminator_table, message_string,
                    text_line(text_index, STRLENGTH(message_string))) ;
        text_index := text_index + STRLENGTH(message_string) ;
        AMP$PUT_NEXT (report_file_identifier, ^text_line,
                      text_index - 1, file_byte_address_x, status_x) ;
        text_index := 1 ;                    { reset index  }
        text_line(1, line_length) := ' ' ;   { blank filler }
        text_index := text_index + 1 ;       { leave column 1 as carriage }
                                             { control character          }
      ELSEIF (text_index + STRLENGTH(message_string) - 1) < line_length
        THEN
        #TRANSLATE (garbage_eliminator_table,
          message_string, text_line(text_index, STRLENGTH(message_string))) ;
        text_index := text_index + STRLENGTH(message_string) ;
        IF new_line_flag
          THEN
          AMP$PUT_NEXT (report_file_identifier, ^text_line,
                        text_index - 1, file_byte_address_x, status_x) ;
          text_index := 1 ;                  { reset index  }
          text_line(1, line_length) := ' ' ; { blank filler }
          text_index := text_index + 1 ;     { leave column 1 as carriage }
                                             { control character          }
        IFEND ;
```

```
      ELSEIF (text_index + STRLENGTH(message_string) - 1) > line_length
        THEN
        string_position_locator := line_length - text_index + 1 ;
        #TRANSLATE (garbage_eliminator_table,
                    message_string(1, string_position_locator),
                    text_line(text_index, string_position_locator)) ;
        text_index := text_index + string_position_locator ;
        AMP$PUT_NEXT (report_file_identifier, ^text_line, text_index - 1,
                      file_byte_address_x, status_x) ;
        text_index := 1 ;                       { reset index  }
        text_line(1, line_length) := ' ' ;      { blank filler }
        text_index := text_index + 1 ;          { leave column 1 as carriage }
                                                { control character          }
        p#put_m (new_line_flag,
                 message_string(string_position_locator + 1, *)) ;
      IFEND ;
    IFEND ;

  PROCEND p#put_m ;


{ ------------------------------------------------------------------------------}
{ This routine looks at the global status variable.  If something has gone     }
{ wrong, then the global error counter is incremented and a formatted message }
{ sent to the error listing file.  To prevent excessive printout, all error   }
{ message reporting is suppressed when the error counter has become too large.}
{ ------------------------------------------------------------------------------}

  PROCEDURE [INLINE] p#inspect_status_variable ;

    IF NOT status.normal
      THEN
      error_count := error_count + 1 ;        { increment error counter    }
      IF error_count < 333
        THEN
        p#display_status_variable ;           { issue the message          }
      ELSEIF error_count = 333
        THEN
        p#put_m (TRUE,
              'Error_Count = 333.  Further message reporting is turned off.') ;
      IFEND ;
    IFEND ;

  PROCEND p#inspect_status_variable ;
```

```
{ ------------------------------------------------------------------------ }
{ The P#DISPLAY_STATUS_VARIABLE routine takes the "raw" status value from the }
{ global storage, formats it by looking up the message template, then appends }
{ the completed diagnostic message onto the report listing file.             }
{ ------------------------------------------------------------------------ }

  PROCEDURE p#display_status_variable ;

    VAR
      annotation : STRING ( 16 ) ,
      message    : OST$STATUS_MESSAGE ,
      line_count : ^ OST$STATUS_MESSAGE_LINE_COUNT ,
      line_size  : ^ OST$STATUS_MESSAGE_LINE_SIZE ,
      line_text  : ^ OST$STATUS_MESSAGE_LINE ,
      pointer    : ^ OST$STATUS_MESSAGE ,
      status_v   : OST$STATUS ;                { local status -- ignored    }

    IF text_index <> 2
      THEN
      p#put_m (TRUE, '') ;             { flush line buffer, put end-of-line }
    IFEND ;

    IF status.NORMAL                           { global status okay ?       }
      THEN
      p#put_m (TRUE, 'NORMAL STATUS') ;
    ELSE
      OSP$FORMAT_MESSAGE (status, OSC$FULL_MESSAGE_LEVEL,
                          line_length - 1 - STRLENGTH(annotation),
                          message, status_v) ;
      IF NOT status_v.NORMAL
      THEN
        p#put_m (TRUE,
       'Error:  Unable to convert status message in
        P#DISPLAY_STATUS_VARIABLE.') ;
      ELSE
        annotation := ' error_status--> ' ;    { first line only            }
        pointer := ^message ;
        RESET pointer ;
        NEXT line_count IN pointer ;
        WHILE line_count^ > 0 DO
          NEXT line_size IN pointer ;
          NEXT line_text : [line_size^] IN pointer ;
          p#put_m (FALSE, annotation) ;
          p#put_m (TRUE, line_text^ (1, line_size^)); { print error
                                                        message text }
          line_count^ := line_count^ - 1 ;
          annotation := '             --> ' ;   { second, third, ... line   }
        WHILEND ;
      IFEND ;
    IFEND ;

  PROCEND p#display_status_variable ;
```

Index

Index

A

AAF$44D_LIBRARY file I-3-1
AAF$DEPENDENCY_FILE I-2-30
AAV$RESOLVE_TIME_LIMIT variable I-2-26
ABANDON_KEY_DEFINITIONS call I-3-4
Access information items I-4-2
ACCESS_MODE attribute I-4-5
Access modes
  Required for each keyed-file interface call I-3-2
  When sharing keyed files I-2-18
Adding records to a sort II-3-5
Adding records to a merge II-3-9
Altering sort key characters II-2-24
Alternate base libraries
  Keyed files I-3-1
  Sort/Merge II-1-1
Alternate index
  Characteristics I-1-16
  Glossary definition A-1
  Information retrieval I-2-38
  Updating I-2-35
Alternate key
  Access information items I-2-36
  Characteristics I-1-15
  Creation I-2-32
    Example I-2-9
  Deletion I-2-32
  Glossary definition A-1
  Selection I-2-33
  Use I-2-33
    Example I-2-49
Alternate key definition
  Description I-1-16
  Glossary definition A-1
AMP$ABANDON_KEY_DEFINITIONS call I-3-4
AMP$APPLY_KEY_DEFINITIONS call I-3-5
AMP$CREATE_KEY_DEFINITION call I-3-7
AMP$CREATE_NESTED_FILE call I-3-14
AMP$DELETE_KEY call I-3-17
AMP$DELETE_KEY_DEFINITIONS call I-3-19
AMP$DELETE_NESTED_FILE call I-3-20
AMP$FETCH_ACCESS_INFORMATION call I-2-36
AMP$GET_KEY call I-3-22
AMP$GET_KEY_DEFINITIONS call I-3-27
AMP$GET_LOCK_KEYED_RECORD call I-3-30
AMP$GET_LOCK_NEXT_KEYED_RECORD call I-3-34
AMP$GET_NESTED_FILE_DEFINITIONS call I-3-38
AMP$GET_NEXT call
  After alternate-key selection I-2-34
  For a keyed file I-2-16
AMP$GET_NEXT_KEY call I-3-40
AMP$GET_NEXT_PRIMARY_KEY_LIST call I-3-43
AMP$GET_PARTIAL call I-2-15
AMP$GET_PRIMARY_KEY_COUNT call I-3-47
AMP$GET_SPACE_USED_FOR_KEY call I-3-51
AMP$LOCK_FILE call I-3-54
AMP$LOCK__KEY call I-3-56
AMP$PUT_KEY call I-3-59
AMP$PUT_NEXT call I-2-10
AMP$PUTREP call I-3-62
AMP$REPLACE_KEY call I-3-64
AMP$REWIND call
  After alternate-key selection I-2-34
  For a keyed file I-2-13
AMP$SELECT_KEY call I-3-66

AMP$SELECT_NESTED_FILE
call I-3-67
AMP$SKIP call
After alternate-key
selection I-2-34
For a keyed file I-2-13
AMP$START call I-3-69
AMP$SYSTEM_HASHING_
PROCEDURE I-1-13
AMP$UNLOCK_FILE call I-3-72
AMP$UNLOCK_KEY call I-3-73
APPLY_KEY_DEFINITIONS
call I-3-5
Ascending sort order A-1
ASCII
Character set B-1
Glossary definition A-1
ASCII6_FOLDED collating
sequence D-11
ASCII6_STRICT collating
sequence D-13
Attribute
Descriptions I-4-5
Settings for new keyed
files I-2-1
AVERAGE_RECORD_LENGTH
attribute I-4-8

B

BEGIN_MERGE_
SPECIFICATION call II-2-4
BEGIN_SORT_SPECIFICATION
call II-2-2
Beginning-of-information A-1
BINARY numeric data
format II-1-8
BINARY_BITS numeric data
format II-1-8
Bit A-1
Block A-1
Block length guideline
attributes I-2-6
BOI A-1
Byte A-2
Byte-addressable file organization
A-2

C

Changing lock intents I-2-25
Character A-2
Character set B-1
Cleared lock I-2-26
Close operation A-2
Close request A-2
COBOL6_FOLDED collating
sequence D-15
COBOL6_STRICT collating
sequence D-17
COLLATE_TABLE
attribute I-4-9
COLLATE_TABLE_NAME
attribute I-4-10
Collated key A-2
COLLATING_ALTER
call II-2-24
COLLATING_CHARACTERS
call II-2-23
COLLATING_NAME call II-2-22
COLLATING_REMAINDER
call II-2-24
Collating sequence A-2
Collation table
Creation D-4
Glossary definition A-2
Listings D-11
Use D-2
Collation weight A-2
Common file structure
attributes I-2-5
Compiling your
program Introduction-2.1
Concatenated key
Description I-1-21
Glossary definition A-2
Concurrent use of keyed
files I-2-18
Condition code Introduction-5
Constant declarations C-1
Content addressing I-1-2
Control-p character I-3-6
Control-t character I-3-6
Conventions used in this
manual 9
*COPYC directives Introduction-1

Copying procedure
decks Introduction-1
CREATE_KEY_DEFINITION
call I-3-7
CREATE_NESTED_FILE
call I-3-14
Creating
Alternate keys I-2-32
Call description I-3-7
Example I-2-49
Keyed file I-2-1
Example I-2-41
Nested file I-3-14
Example I-2-55
CYBIL
Constant declarations C-1
Manual set 8
Object libraries
Introduction-2.1
Procedure declarations
Introduction-1
Type declarations C-1

D

Data block
Description I-1-3
Glossary definition A-3
Data-block split
Description I-1-4
Glossary definition A-3
DATA_PADDING
attribute I-4-11
Deadlock I-2-29
Deck
Glossary definition A-3
Names Introduction-1
Declarations
Constant and type C-1
Procedure Introduction-1
Default value A-3
DEFINE_USER_COLLATING_
TABLE call II-2-11
DELETE_KEY call I-3-17
DELETE_KEY_DEFINITION
call I-3-19
DELETE_NESTED_FILE
call I-3-20
Deleting
Alternate keys I-2-32
Keyed-file records I-3-17
Nested files I-3-20
Records from a sort or
merge II-3-9
Descending sort order A-3
Direct-access file
Attributes I-2-7
Creation I-2-1
Glossary definition A-3
Hashing procedure I-1-13
Organization I-1-10
Positioning I-2-13
Primary key I-1-14
Re-creation I-2-11
Structure I-1-10
Discarding alternate-key definition
and deletion requests I-3-4
DISPLAY63_FOLDED collating
sequence D-19
DISPLAY63_STRICT collating
sequence D-21
DISPLAY64_FOLDED collating
sequence D-23
DISPLAY64_STRICT collating
sequence D-25
Duplicate key value
Description I-1-17
Glossary definition A-3
DUPLICATE_VALUE_
INSERTED item I-4-12

E

EBCDIC
Glossary definition A-3
Collating sequence D-27
EBCDIC6_FOLDED collating
sequence D-33
EBCDIC6_STRICT collating
sequence D-35
Embedded key A-3
EMBEDDED_KEY attribute
I-4-12

Empty block chain I-3-17
End-of-information A-4
End_of_key_list position I-2-37
END_SPECIFICATION
call II-2-29
Entry point A-4
EOI A-4
EOI_BYTE_ADDRESS
item I-4-13
Equal sort key processing
Owncode procedure 5 II-3-12
SMP$RETAIN_ORIGINAL_
ORDER call II-2-21
SMP$SUM call II-2-26
ERROR_COUNT item I-4-13
ERROR_EXIT_NAME
attribute I-4-14
ERROR_EXIT_PROCEDURE
attribute I-4-15
Error exit procedure
use Introduction-4
ERROR_FILE call II-2-10
ERROR_LEVEL call II-2-11
ERROR_LIMIT attribute
Description I-4-15
Error limit processing for
duplicate key values I-1-18
ERROR_STATUS item I-4-16
ESTIMATED_NUMBER_
RECORDS call II-2-15
ESTIMATED_RECORD_COUNT
attribute I-4-16
Example
Creating an alternate
key I-2-49
Creating an indexed-sequential
file I-2-41
Creating and deleting nested
files I-2-55
Sort/Merge owncode
procedure II-3-13
Sort/Merge
specification II-1-14
Updating an indexed-sequential
file I-2-45
Exception condition
Introduction-4
Exception records file A-4
EXCEPTION_RECORDS_FILE
call II-2-16
Exclusive_Access lock
intent II-2-24
Executing your program
Introduction-2.1
Expanding your program
Introduction-2
Expired lock
Conditions I-2-28
Description I-2-26
External reference A-4

F

F record type A-4
FETCH_ACCESS_
INFORMATION call I-2-36
Fetching
Access information
items I-2-36
Alternate index
information I-2-38
Field A-4
FIFO order I-1-17
File A-4
File access modes I-3-2
File attribute (see Attribute)
File cycle A-4
FILE_LENGTH attribute I-4-16
FILE_LIMIT attribute I-4-17
File lock
Clearing I-3-72
Description I-2-30
Request I-3-54
File organization A-4
FILE_ORGANIZATION
attribute I-4-17
File position
After alternate-key
selection I-2-37
Glossary definition A-4
FILE_POSITION item I-4-18
File reference A-5
File structure attributes I-2-4

First-in-first-out order I-1-17
Fixed record length attribute I-2-2
Floating sign numeric data
format II-1-8
Flush request A-5
Flushing A-5
FORCED_WRITE attribute
Description I-4-19
When the keyed file is
shared I-2-5
FROM_FILES call II-2-4

G

GET_KEY call I-3-22
GET_KEY_DEFINITIONS
call I-3-27
GET_LOCK_KEYED_RECORD
call I-3-30
GET_LOCK_NEXT_KEYED_
RECORD call I-3-34
GET_NESTED_FILE_
DEFINITIONS call I-3-38
GET_NEXT call
After alternate-key
selection I-2-34
For a keyed file I-2-16
GET_NEXT_KEY call I-3-40
GET_NEXT_PRIMARY_KEY_
LIST call I-3-43
GET_PARTIAL call I-2-15
GET_PRIMARY_KEY_COUNT
call
Description I-3-47
Processing I-2-38
Getting
Alternate-key definitions I-3-27
Index space used for key value
range I-3-51
Keyed-file records I-2-16
Nested-file definitions I-3-38
Primary-key value count I-3-47
GLOBAL_ACCESS_MODE
attribute I-4-20
GLOBAL_FILE_NAME attribute
I-4-20
GLOBAL_SHARE_MODE
attribute I-4-21

H

Hashing procedure
Attribute I-2-7
Description I-1-13
Glossary definition A-5
HASHING PROCEDURE NAME
attribute I-4-21
Home block
Attribute I-2-7
Description I-1-10
Glossary definition A-5

I

Index block
Description I-1-3
Glossary definition A-5
Index-block split
Description I-1-6
Glossary definition A-5
Index level
Description I-1-6
Glossary definition A-5
INDEX_LEVELS attribute I-4-22
Index level overflow
Description I-1-6
Glossary definition A-6
INDEX_PADDING
attribute I-4-22
Index record A-6
Indexed-sequential file
Attributes I-2-7
Creation I-2-1
Glossary definition A-6
Organization I-1-2
Re-creation I-2-10
Structure I-1-2
Initial home block count
Attribute I-2-7; I-4-23
Description I-1-11
Instance of open A-6
INTEGER numeric data
format II-1-8
INTEGER_BITS numeric data
format II-1-8
Integer key A-6
Invalid sort records II-1-13

J

Job A-6

K

Key A-6
KEY call II-2-9
Key count I-3-47
KEY_LENGTH attribute I-4-23
Key list
  Description I-1-17
  Glossary definition A-6
KEY_POSITION attribute I-4-23
Key relation positioning I-2-14
Key type
  Glossary definition A-7
  Keyed-file attribute I-2-4
  Sort/Merge II-1-5
KEY_TYPE attribute I-4-24
Keyed-file
  Attribute
    Descriptions I-4-5
    Setting for a new file I-2-1
  Calls I-3-1
  Concepts I-1-1
  Creation I-2-1
  Organization I-1-1
    Glossary definition A-7
  Positioning I-2-13
  Reading I-2-15
  Records I-2-1
  Sharing I-2-18
  Writing I-2-10
  Use I-2-12
Keyed-file interface object library I-3-1

L

LAST_ACCESS_OPERATION item I-4-25
LAST_OP_STATUS item I-4-27
LEVELS_OF_INDEXING item I-4-27
Library A-7
LIST_FILE call II-2-17
LIST_OPTION call II-2-18
LOAD_COLLATING_TABLE call II-2-18.2
Local file A-7
Local file name A-7
Local path A-7
Lock
  Clearing I-2-26
  Deadlock I-2-29
  Effect on calls I-2-31
  Expiration I-2-26
  Expiration conditions I-2-28
  Intent
    File locks I-2-30
    Key locks I-2-24
    Switching I-2-25
  Maximum I-2-30
  Processing I-2-30
  Timeout period I-2-26
  Waiting I-2-26
Lock expiration time
  Attribute I-2-9; I-4-27
  Use I-2-27
Lock file I-2-30
LOCK_FILE call I-3-54
LOCK__KEY call I-3-56
Lock manager I-2-21

M

$MAIN FILE I-2-24
Major key
  Glossary definition A-8
  Positioning of a keyed file I-2-14
Major sort key A-8
Mass storage A-8
MAX_BLOCK_LENGTH attribute I-4-28
MAX_RECORD_LENGTH attribute I-4-28
Merge A-8
Merge input record order II-2-29
MESSAGE_CONTROL attribute I-4-29
MIN_RECORD_LENGTH attribute I-4-30

Minor sort key A-8
Module A-8

N

Naming
convention Introduction-6
Nested file
Creation I-3-14
Definition record I-3-14
Deletion I-3-20
Descripton I-1-24
Example I-2-54
Glossary definition A-8
Nonembedded key
Description I-1-9
Glossary definition A-8
NOS/VE predefined collation table
(see Predefined collation table)
NULL_ATTRIBUTE attribute
I-4-30
NULL_ITEM item I-4-30
Null suppression
Description I-1-19
Glossary definition A-8
Null values I-1-19
NUMBER_OF_NESTED_FILES
item I-4-31
Numeric data formats II-1-7
NUMERIC_FS numeric data
format II-1-8
NUMERIC_LO numeric data
format II-1-8
NUMERIC_LS numeric data
format II-1-8
NUMERIC_NS numeric data
format II-1-9
NUMERIC_TO numeric data
format II-1-9
NUMERIC_TS numeric data
format II-1-9

O

Object code A-9
Object library A-9
Open operation A-9
Open request A-9
OPEN_POSITION
attribute I-4-31
Ordered by primary key I-1-17
Ordering manuals 10
OST$STATUS record
Introduction-4
OSV$ASCII6_FOLDED collating
sequence D-11
OSV$ASCII6_STRICT collating
sequence D-13
OSV$COBOL6_FOLDED
collating sequence D-15
OSV$COBOL6_STRICT collating
sequence D-17
OSV$DISPLAY63_FOLDED
collating sequence D-19
OSV$DISPLAY63_STRICT
collating sequence D-21
OSV$DISPLAY64_FOLDED
collating sequence D-23
OSV$DISPLAY64_STRICT
collating sequence D-25
OSV$EBCDIC collating
sequence D-27
OSV$EBCDIC6_FOLDED
collating sequence D-33
OSV$EBCDIC6_STRICT collating
sequence D-35
Overflow block
Description I-1-10
Glossary definition A-9
Overpunch signed numeric
data II-1-10
Owncode A-9
OWNCODE_FIXED_RECORD_
LENGTH call II-2-18.4
OWNCODE_MAX_RECORD_
LENGTH call II-2-19
Owncode procedure
Parameters II-3-2
Processing II-3-1
Specification II-3-1
OWNCODE_PROCEDURE_n
call II-2-20
Owncode 1 procedure
Processing II-3-5

Specification II-2-20
Owncode 2 procedure
Processing II-3-7
Specification II-2-20
Owncode 3 procedure
Processing II-3-9
Specification II-2-20
Owncode 4 procedure
Processing II-3-11
Specification II-2-20
Owncode 5 procedure
Processing II-3-12
Specification II-2-20

P

PACKED numeric data format II-1-9
PACKED_NS numeric data format II-1-9
Padding
Description I-1-4
Glossary definition A-9
Path A-9
Pause_break character I-3-5
Permanent file A-9
PERMANENT_FILE attribute I-4-32
Piece
Description I-2-21
Glossary definition A-9
Positioning
Keyed files I-2-13
Using alternate keys I-2-34
Predefined collation table
Listings D-11
Use D-2
Preserve_Access_and_Content lock intent I-2-24
Preserve_Content lock intent I-2-24
Primary key
Attributes I-2-3
Characteristics
Direct-access I-1-14
Indexed-sequential I-1-9
Glossary definition A-10
PRIMARY_KEY item I-4-32
Primary-key-value order I-1-17
Procedure call use Introduction-1
Procedure calls
Keyed-file interface I-3-1
Sort/Merge II-2-1
Procedure deck names Introduction-1
Process identifiers Introduction-5
Processing attributes I-2-8
Processing a keyed file I-2-12
Program examples
Keyed-file interface I-2-40
Sort/Merge interface II-1-14
Program-library list A-10
PUT_KEY call I-3-59
PUT_NEXT call I-2-10
PUTREP call I-3-62
Putting keyed-file records I-2-10

R

Random access
Description I-2-17
Glossary definition A-10
Reading
Keyed files I-2-15
Using alternate keys I-2-34
REAL numeric data format II-1-9
Reca owncode parameter II-2-2
Recb owncode parameter II-2-2
Record A-10
Record attributes I-2-2
Record length
Keyed-files I-2-2
Sort/Merge II-1-12
RECORD_LIMIT attribute I-4-32
RECORD_TYPE attribute I-4-33
RECORDS_PER_BLOCK attribute I-4-33
Re-creating a keyed file I-2-10
Remainder collation step II-2-24
Repeating groups
Description I-1-22
Glossary definition A-10
REPLACE_KEY call I-3-64
Replacing keyed-file records I-3-64

RESIDUAL_SKIP_COUNT
item I-4-34
Result array II-2-3
RETAIN_ORIGINAL_ORDER
call II-2-21
Return_code owncode
parameter II-3-2
RETURN_OPTION
attribute I-4-34
Rewind A-10
REWIND call
After alternate-key
selection I-2-34
For a keyed file I-2-13
Ring A-10
RING_ATTRIBUTES attribute
Description I-4-35
For a hashing procedure I-1-13
Rla owncode parameter II-3-2
Rlb owncode parameter II-3-2

S

SCL A-10
SCU Introduction-1
SELECT_KEY call I-3-60
SELECT_NESTED_FILE
call I-3-67
SELECTED_KEY_NAME
item I-4-35
SELECTED_NESTED_FILE
item I-4-36
Selecting a key I-2-33
Self-deadlock condition I-2-29
Sequential access
Direct-access file I-2-16
Glossary definition A-11
Indexed-sequential file I-2-15
Sequential file organization A-11
Setting file attributes I-2-1
Sharing keyed files I-2-18
Short sort records II-1-13
Sign overpunch representation
II-1-11
Signed numeric sort data II-1-10
SKIP call
After alternate-key selection
I-2-34
For a keyed file I-2-13
SMF$LIBRARY file II-1-2
SMP$BEGIN_MERGE_
SPECIFICATION call II-2-4
SMP$BEGIN_SORT_
SPECIFICATION call II-2-2
SMP$COLLATING_ALTER
call II-2-24
SMP$COLLATING_
CHARACTERS call II-2-23
SMP$COLLATING_NAME
call II-2-22
SMP$COLLATING_
REMAINDER call II-2-24
SMP$DEFINE_USER_
COLLATING_TABLE
call II-2-11
SMP$END_SPECIFICATION
call II-2-29
SMP$ERROR_FILE call II-2-12
SMP$ERROR_LEVEL call
II-2-13
SMP$ESTIMATED_NUMBER_
RECORDS call II-2-15
SMP$EXCEPTION_RECORDS_
FILE call II-2-16
SMP$FROM_FILES call II-2-5
SMP$KEY call II-2-9
SMP$LIST_FILE call II-2-17
SMP$LIST_OPTION call II-2-18
SMP$LOAD_COLLATING_
TABLE call II-2-18.2
SMP$OWNCODE_FIXED_
RECORD_LENGTH
call II-2-18.4
SMP$OWNCODE_MAX_
RECORD_LENGTH call II-2-19
SMP$OWNCODE_
PROCEDURE_n call II-2-20
SMP$RETAIN_ORIGINAL_
ORDER call II-2-21
SMP$STATUS call II-2-25
SMP$SUM call II-2-26
SMP$TO_FILE call II-2-7
SMP$VERIFY call II-2-29
Sort A-11
Sort key
Description II-1-3

Glossary definition A-11
Sort order
Description II-1-12
Glossary definition A-11
Sort/Merge
Call order II-2-1
Error levels II-2-13
Example program II-1-14
Input files II-2-5
Object library II-1-2
Output file II-2-7
Owncode procedure processing II-3-1
Record length II-1-12
Record insertion II-3-5
Record deletion II-3-9
Statistics II-2-3
Valid records II-1-13
Source code A-11
Source Code Utility Introduction-1
Source library A-11
Sparse-key control
Description I-1-20
Glossary definition A-11
START call I-3-69
Statistics A-11
STATUS call II-2-25
Status checking
Description Introduction-4
Procedures E-1
Status record contents Introduction-4
Status variable A-11
Submitting comments 10
SUM call II-2-26
Sum fields A-12
Summing A-12
Switching lock intents I-2-25
System Command Language A-12
System hashing procedure I-1-13
System naming convention Introduction-6

T

Task A-12
Terminate_break character I-3-5
Timeout period I-2-26
TO_FILE call II-2-7
Trivial-error limit
Attribute description I-4-15
Processing duplicate-key value errors I-1-18
Type checking Introduction-3
Type declarations C-1

U

U record type A-12
Uncollated key A-12
UNLOCK ALL call I-3-73
UNLOCK_FILE call I-3-72
UNLOCK_KEY call I-3-73
Updating an alternate index I-2-35
Using
Alternate keys I-2-33
Example I-2-49
Keyed files I-2-12
Example I-2-45

V

V record type A-12
Validating sort data II-1-13
VERIFY call II-2-29

W

Waiting for a lock I-2-26
Working storage area A-12
Writing
After alternate-key selection I-2-35
Keyed-file records I-2-10

Z

Zero-length sort records II-1-13

CYBIL for NOS/VE Keyed-File and Sort/Merge Interfaces 60464117 B

We would like your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

Who Are You?	How Do You Use This Manual?	Which Do You Also Have?
☐ Manager	☐ As an Overview	☐ Any SCL Manuals
☐ Systems Analyst or Programmer	☐ To Learn the Product/System	☐ CYBIL Language Definition
☐ Applications Programmer	☐ For Comprehensive Reference	☐ CYBIL System Interface
☐ Operator	☐ For Quick Look-up	☐ CYBIL File Management
☐ Other _____		☐ CYBIL Sequential and Byte-Addressable Files

What programming languages do you use? _____

Which are helpful to you? ☐ Procedures Index (inside covers) ☐ Glossary ☐ Related Manuals page ☐ Character Set ☐ Other: _____

How Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple ☐ Too complex)
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? ☐ Yes ☐ No

Continue on other side

From:

Name _____ Company _____

Address _____ Date _____

_____ Phone No. _____

_____

Please send program listing and output if applicable to your comment.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 8241 MINNEAPOLIS, MN.

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA

Publications and Graphics Division
Mail Stop: SVL104
P.O. Box 3492
Sunnyvale, California 94088-3492

FOLD FOLD

Comments (continued from other side)